C#

Pearson Education

ADDISON-WESLEY

Jürgen Bayer

C#

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich

Umwelthinweis:
Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.
Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

10 9 8 7 6 5 4 3 2 1

06 05 04 03 02

ISBN 3-8273-1856-4

Einbandgestaltung:	Vera Zimmermann, Mainz
Lektorat:	Frank Eller, feller@pearson.de
Korrektorat:	Barbara Decker, München
Herstellung:	Philipp Burkart, pburkart@pearson.de
Satz und Layout:	mediaService, Siegen (www.media-service.tv)
Druck und Verarbeitung:	Nørhaven, Viborg (DK)

Printed in Denmark

Inhaltsverzeichnis

Teil III – Go ahead!

Vorwort

Dieses Buch beschreibt die Sprache C# (gesprochen als »CSharp«) und die für die tägliche Arbeit wichtigen Aspekte des zugrunde liegenden .NET-Frameworks. Es behandelt im Wesentlichen die Windows-Programmierung, beschreibt aber auch, wie Sie mit Datenbanken arbeiten.

Das Buch legt Wert auf die Grundlagen der Sprache und die Lösung alltäglicher Programmierprobleme. Es setzt nicht voraus, dass Sie bereits in C# oder einer ähnlichen Programmiersprache entwickeln. Als allgemeine Programmiereinführung können Sie das Buch allerdings nicht verwenden. Die einzelnen Kapitel setzen voraus, dass Sie grundlegende Programmierkenntnisse besitzen (die Sie sich bei Bedarf über das Buch »Programmieren« aus der Nitty-Gritty-Reihe aneignen können). Da C# eine objektorientierte Programmiersprache ist, sind Kenntnisse der objektorientierten Programmierung (OOP) von Vorteil. Wenn Sie diese nicht besitzen, können Sie bei Bedarf (wenn im Buch das eine oder andere OOP-Thema vielleicht etwas knapp beschrieben wird) den Online-Artikel »OOP-Grundlagen« lesen, den Sie auf der Website zum Buch finden.

Beim Schreiben dieses Buchs hatte ich den Leitgedanken, dass ein Programmierer im Buch die wichtigen Informationen findet, die er zur sinnvollen und sicheren Softwareentwicklung mit C# benötigt. Randinformationen, die Sie für die Programmierung eigentlich nicht benötigen, finden Sie also nur sehr selten.

Ich danke meínem Lektor Frank Eller für die nette und verständnisvolle Betreuung während der Manuskripterstellung. Ganz besonders möchte ich mich bei Thomas Herold bedanken, dessen Anregungen diesem Buch den letzten Schliff gaben. Daneben danke ich all den Autoren der genialen (literarischen) Bücher, deren Protagonisten ich in viele Beispiele meines Buchs übernommen habe. Schließlich möchte ich kurz auch noch meinem Computer dafür danken, dass er während der Manuskriptfertigung nie abgestürzt ist (das muss ja auch mal sein).

Viel Spaß mit C#

Jürgen Bayer
juergen.bayer@addison-wesley.de

Teil I

Start up!

1 Einführung

1.1 Zum Buch

1.1.1 Die .NET-Version, auf der dieses Buch basiert

Dieses Buch wurde mit der Version RC[1] 3, Build 9412 geschrieben. Diese Version ist laut Microsoft »Feature complete«, enthält also alle Features des ersten Release (das Anfang 2002 erscheinen wird), und läuft sehr stabil. Für das erste Release erwarte ich eigentlich keine Änderungen, die dieses Buch betreffen. Falls Anpassungen im Buch notwendig sind, finden Sie auf der Website zum Buch dann einen entsprechenden Artikel.

1.1.2 Die Buchseite bei www.nitty-gritty.de

Über die Internet-Startseite von Nitty-Gritty (*www.nitty-gritty.de*) erreichen Sie über den BÜCHER-Link die Seiten zu den einzelnen Nitty-Gritty-Büchern. Auf der Seite zum C#-Buch finden Sie zusätzliche Artikel (deren Themen nicht mehr ins Buch gepasst haben) und alle Beispiele, die ich für das Buch erarbeitet habe. Die Beispiele sind in eine einzige Datei gepackt. Entpacken Sie die Datei in einen Ordner Ihrer Wahl. Achten Sie aber darauf, dass Ihr Entpack-Programm die Ordnerstruktur des Archivs beibehält.

1.1.3 Was nicht im Buch behandelt wird

Das Buch besitzt nur etwa 450 Seiten. Mehr war (leider) nicht möglich. Deshalb konnte ich nicht alle Themen behandeln, die ich gerne besprochen hätte. Einige wichtige Themen, wie die Grundlagen der Sprache, die OOP, die Windowsprogrammierung und der Datenbankzugriff sind so umfangreich, dass diese recht viel Platz im Buch beanspruchen. Es war für mich natürlich sehr schwer die Themen auszuwählen, die für Sie interessant sind. Ich habe aber die – aus meiner Sicht – die für die Praxis wichtigen Themen ausgewählt. Um diese so zu besprechen, dass möglichst wenig offene Fragen übrig bleiben,

1. Release Candidate. Das ist ein Kandidat für das erste Release (die Version 1.0), der der letzten Betaversion folgt.

konnte ich andere Themen leider erst gar nicht aufnehmen. Die wichtigsten fehlenden Themen zeigt die folgende Auflistung:

- Internetprogrammierung: Die Internetprogrammierung wird recht ausführlich im Internetprogrammierung-Buch der Nitty-Gritty-Reihe behandelt.
- Remoting: Sehr interessantes Thema: Hierüber können Sie eine Komponente auf einem Rechner laufen lassen und von entfernten Rechnern aus verwenden.
- Zugriff auf das Dateisystem: Das Lesen und Schreiben von Textdateien wäre beispielsweise interessant gewesen.
- Umgang mit XML-Dateien: XML wird im Buch zwar kurz behandelt, nicht aber der teilweise komplexe Umgang mit den mächtigen XML-Features.
- Threading: Die Möglichkeit, Teile eines Programms in einzelnen Threads (»Programmfäden«) parallel ablaufen zu lassen ist sehr interessant.
- Grafikprogrammierung: Ebenfalls ein sehr interessantes Thema.
- Erzeugen von eigenen Steuerelementen: Die Möglichkeit, unter .NET Steuerelemente von vorhandenen Steuerelementen abzuleiten und in der Funktionsweise oder im Aussehen zu verändern ist für die Praxis zwar zunächst nicht allzu wichtig, aber trotzdem sehr interessant.
- Windows-API-Programmierung: Eigentlich sollte das Windows-API unter .NET gar nicht mehr verwendet werden. Einige wenige Dinge müssen aber wohl immer noch über Windows-API-Funktionen programmiert werden.
- Drucken: Das .NET-Framework stellt Ihnen Klassen zur Verfügung, über die das Drucken recht einfach wird.
- Datenbankberichte mit Crystal Reports: Der als allgemeiner Standard für das Erstellen von Berichten aus Datenbanken anerkannte Berichtsgenerator Crystal Reports ist in .NET integriert. Interessant.

Wir denken gerade darüber nach, in der Nitty-Gritty-Reihe ein separates C#-Buch zu veröffentlichen, das diese (und noch andere) Themen

behandelt, quasi als Teil 2 dieses Buchs. Viele dieser Themen finden Sie aber auch im Internet auf den Webseiten, die ich auf der Seite 20 vorstelle.

1.1.4 Typografische Konventionen

Dieses Buch verwendet einige typografische Konventionen, die jedoch nicht allzu sehr vom allgemeinen Standard abweichen.

Syntaxbeschreibungen

Wenn ich die Syntax einer Deklaration beschreibe, verwende ich eine kompakte und übersichtliche Form, wie im folgenden Beispiel:

```
int IndexOf({string | char} value
  [, int startIndex][, int count])
```

Normale Wörter sind sprachspezifische Schlüsselwörter. Kursive Wörter sind Platzhalter für Eingaben, die Sie spezifizieren müssen. Die in eckigen Klammern stehenden Elemente sind optional. Diese Elemente können Sie, müssen Sie jedoch nicht angeben. Die eckigen Klammern sind nur Teil der Syntaxbeschreibung und werden nicht im Programmcode angegeben. Wenn an einer Stelle mehrere Varianten möglich sind, werden diese in der Syntaxbeschreibung durch ein | (das Oder-Zeichen in C#) voneinander getrennt. Handelt es sich dabei um eine nicht optionale Angabe, schließe ich diese in geschweifte Klammern ein. Diese Klammern werden dabei natürlich auch nicht mit angegeben.

Sie müssen bei den Syntaxbeschreibungen ein wenig aufpassen: Wenn Sie mit Arrays arbeiten, gehören die eckigen und die geschweiften Klammern zur Syntax. Geschweifte Klammern werden außerdem auch für das blockweise Zusammenfassen von Anweisungen eingesetzt. Sie müssen die Syntaxbeschreibungsklammern also von den sprachspezifischen Klammern unterscheiden. Leider blieb mir keine andere Möglichkeit, die teilweise umfangreichen Deklarationen der Methoden mancher Klassen übersichtlich und kompakt darzustellen.

Beispiel-Listings

Beispiel-Listings werden in der Schriftart *Courier* dargestellt:

```
string now;
now = DateTime.Now.ToShortDateString();
MessageBox.Show("Heute ist der " + now);
```

Typografische Konventionen im Fließtext

Im normalen Text werden *sprachspezifische Schlüsselwörter* in der Schriftart `Courier` dargestellt. Wörter in Kapitälchen im normalen Text bezeichnen Teile der Benutzerschnittstelle, wie z. B. Menübefehle und Schalter. Menübefehle, die in einem Menü untergeordnet sind, werden mit den übergeordneten Menüs angegeben, wobei die einzelnen Menüebenen durch einen Schrägstrich getrennt werden (z. B. DATEI / BEENDEN). Datei- und Ordnernamen werden kursiv formatiert. Internetadressen werden folgendermaßen gekennzeichnet: *www.nitty-gritty.de*. Tastenkappen wie F1 stehen für Tasten und Tastenkombinationen, die Sie betätigen können, um bestimmte Aktionen zu starten.

1.2 Installation

Vorüberlegungen

Zur Ausführung von C#-Programmen ist das .NET-Framework notwendig. Dieses müssen Sie also auf jeden Fall installieren. Wenn Sie Visual Studio besitzen, sollten Sie diese Entwicklungsumgebung ebenfalls installieren. Mit Visual Studio ist die Entwicklung von .NET-Anwendungen wesentlich einfacher als mit anderen Entwicklungsumgebungen. Alternativ können Sie Ihre Programme auch in einem einfachen Editor oder einem speziellen (Freeware-).NET-Editor (z. B. Sharp Develop) entwickeln und über den Kommandozeilen-Compiler des .NET-Framework kompilieren. Dann reicht das .NET-Framework SDK (Software Development Kit) vollkommen aus. In Kapitel 2 zeige ich, wie Sie C#-Programme entwickeln und kompilieren.

Nach meinen Erfahrungen lässt sich das .NET-Framework problemlos installieren, auch wenn das Betriebssystem bereits längere Zeit in Betrieb ist (und folglich viele Installationen zuvor erfolgt sind). Für Visual Studio gilt das allerdings nicht unbedingt. Bei diversen Versuchen ließ sich Visual Studio nur dann problemlos installieren, wenn das Betriebssystem frisch war. Versuche mit bereits länger laufenden Systemen resultierten meist in einigen Fehlermeldungen und dem Fehlschlagen der Installation. Da diese leider erst am Ende der Installation gemeldet werden, nachdem das Setup den alten Zustand in einer zeitaufwändigen Aktion wiederhergestellt hat, verlieren Sie sehr viel Zeit, wenn Sie die Installation trotzdem ausprobieren. Ich rate Ihnen also dringend dazu, Windows vor Visual Studio komplett neu zu installieren.

Betriebssystem und Speicheranforderungen

Das .NET-Framework lässt sich unter Windows 98, Me, NT 4, 2000 und XP installieren und benötigt ca. 200 MB freien Festplattenplatz. Besondere Speicheranforderungen bestehen scheinbar nicht.

Visual Studio kann nur unter Windows NT 4, 2000 oder XP installiert werden. Arbeitsspeicher sollte reichlich vorhanden sein. Microsoft empfiehlt 128 bis 256 MB, je nach Betriebssystem. Die Anforderungen an den freien Festplattenplatz sind enorm. Auf der Systempartition müssen 500 MB frei sein, die Partition, auf der Visual Studio installiert wird, sollte (laut Microsoft) 3 GB freien Platz besitzen. Nach meinen Erfahrungen reichen aber auch ca. 2 GB freier Platz vollkommen aus.

Vorinstallationen

Vor der Installation des .NET-Framework oder von Visual Studio.NET sollten Sie die Internet-Informationsdienste bzw. den IIS (Internet Information Server) installieren. Diese Dienste werden benötigt, wenn Sie Internetanwendungen erzeugen und testen wollen. Eine nachträgliche Installation ist schwierig, da das Installationsprogramm einige wichtige Komponenten nicht installiert, wenn die Internet-Informationsdienste fehlen.

Wenn Sie die Internet-Informationsdienste bzw. den IIS auf Ihrem System ausführen, ist Ihr Rechner ein potenzieller Angriffspunkt für Viren und Hackerangriffe. Unterschätzen Sie das nicht: Wenn diese Dienste auf meinem Rechner laufen, während ich mit dem Internet verbunden bin, meldet meine Firewall wesentlich mehr Angriffe als sonst. Trotz Firewall hatte ich schon mehrfach den Nimda-Virus auf meinem Rechner, wenn die IIS-Dienste liefen. Installieren Sie das Security Toolkit, das Sie bei www.microsoft.com/security finden, um Ihren Rechner wenigstens so gut es geht abzusichern.

Unter Windows NT 4 installieren Sie den IIS über das Windows NT Option Pack, das Sie auf neueren NT-Installations-CDs, auf Service Pack-CDs oder (wohl eher) im Internet unter der Adresse *www.microsoft.com/msdownload/ntoptionpack/askwiz.asp* finden. Achten Sie darauf, dass Sie die für Ihr System korrekte Variante herunterladen. Vor dem Option Pack müssen Sie den Internet Explorer ab Version 4.01 und das NT Service Pack ab Version 3 installieren.

Unter Windows 2000 und XP installieren Sie die Internet Informationsdienste über die Windows-Komponenten. Öffnen Sie die Systemsteuerungen, wählen Sie den Eintrag SOFTWARE und klicken Sie dann auf WINDOWS-KOMPONENTEN HINZUFÜGEN/ENTFERNEN. Installieren Sie dort die Internet Informationsdienste, idealerweise mit allen Unteroptionen.

Das .NET-Framework erfordert zudem noch die Installation des Internet Explorers ab Version 5.01 und von MDAC (Microsoft Data Access Components) ab Version 2.6. Den Internet Explorer finden Sie im Internet unter der Adresse *www.microsoft.com/windows/ie*. MDAC können Sie bei *www.microsoft.com/data* downloaden. Beachten Sie, dass diese Installationen nicht notwendig sind, wenn Sie Visual Studio installieren.

Visual Studio

Wenn Sie Visual Studio installieren, wird automatisch das eventuell notwendige Windows-Komponenten-Update installiert, das auch das .NET-Framework SDK enthält. Zur Installation von Visual Studio

legen Sie einfach die erste CD des Installationspakets ein. Rufen Sie *Setup.Exe* auf, falls die Installation nicht automatisch startet. Das Installationsprogramm fordert Sie dann auf, die CD mit dem Update der Windows-Komponenten einzulegen, die Bestandteil des Pakets ist. Nach dem Update dieser Komponenten werden Sie erneut aufgefordert, die erste Installations-CD einzulegen. Die Installation läuft meist problemlos ab (zumindest dann, wenn Windows noch frisch ist), benötigt aber einige Zeit.

Das .NET-Framework SDK

Wenn Sie nicht im Besitz von Visual Studio sind, sollten Sie das .NET-Framework SDK installieren. Das .NET-Framework SDK enthält neben dem .NET-Framework verschiedene Tools, die für die Programmentwicklung notwendig sind. Das aktuelle .NET-Framework finden Sie unter der Adresse *msdn.microsoft.com/net*. Klicken Sie hier auf den Link .NET Framework SDK und wählen Sie dann das .NET Framework Full SDK, das auch einige für Entwickler wichtige Tools enthält. Das Setup ist mehr als 100 MB groß, gehen Sie während des Download also irgendwo einen Kaffee trinken. In neuen Windows-Versionen wird das .NET-Framework übrigens Bestandteil des Betriebssystems sein. Das SDK müssen Sie dann allerdings wahrscheinlich trotzdem herunterladen und installieren.

Starten Sie zur Installation einfach die Datei *Setup.exe*. Die Installation ist recht einfach, für die wenigen Optionen können Sie die Voreinstellungen übernehmen. Das .NET-Framework-SDK benötigt ca. 200 MB auf der Festplatte.

1.3 Wo Sie Hilfe und weitere Informationen finden

Die .NET-Framework-Dokumentation

Die äußerst umfangreiche .NET-Framework-Dokumentation finden Sie im Startmenü von Windows im Ordner Programme / Microsoft .NET Framework SDK. Klicken Sie dort auf den Eintrag Documentation. Unter .NET Framework SDK / .NET Framework Reference / .NET Framework Class Library finden Sie dann die Referenz der Klassen des .NET-Framework. Unter .NET Framework SDK / .NET Framework Refe-

RENCE / .NET FRAMEWORK COMPILER AND LANGUAGE REFERENCE / C# LANGUAGE SPECIFICATION FINDEN SIE DIE C#-REFERENZ.

Diese Dokumentationen können Sie innerhalb von Visual Studio auch kontextsensitiv nutzen, indem Sie den Eingabecursor auf einen Bezeichner in Ihrem Quelltext setzen und dann in der dynamischen Hilfe (im Fenster unten rechts) ein passendes Hilfethema auswählen.

Die Quickstart-Tutorials

In den Quickstart-Tutorials finden Sie viele, einfach aufgebaute und informative Artikel mit guten Beispielen zu den verschiedenen .NET-Features. Die Quickstart-Tutorials können Sie auf Ihrem Rechner installieren oder direkt im Internet abrufen (*www.gotdotnet.com/quickstart/default.aspx*). Wenn Sie diese Artikel auf Ihrem Rechner verwenden wollen, muss allerdings der IIS laufen (weil die Artikel auf dem lokalen Webserver installiert werden). Falls dies nicht der Fall ist, können Sie sich die Installation sparen.

Wenn Sie den Eintrag SAMPLES AND QUICKSTART TUTORIALS im Startmenüeintrag des .NET-Framework-SDK anklicken, wird zunächst eine Seite angezeigt, über die Sie die Installation starten können. Gehen Sie beide Schritte durch. Im ersten Schritt installieren Sie eine SQL-Server-Instanz und einige Beispieldatenbanken. Den SQL-Server können Sie auch gut für die Datenbankbeispiele in Kapitel 8 einsetzen. Im zweiten Schritt installieren Sie dann die Tutorials.

Suchen im Internet

Wenn Sie einmal bei der Programmierung das eine oder andere Problem haben, können Sie dieses (natürlich) meist über das Internet lösen. Eine der ersten Adressen ist dann Microsoft. Microsoft stellt eine sehr große Anzahl an Informationen in seinem MDSN (Microsoft Developer Network) zur Verfügung. Auf der MSDN-Suchseite, die Sie bei *search.microsoft.com/search/us/dev/default.asp* finden, können Sie sehr flexibel nach Problemlösungen oder allgemeinen Informationen suchen. Achten Sie darauf, dass Sie im Suchbegriff immer ».NET« oder »CSharp« angeben, damit Sie nur Artikel finden, die etwas mit C# oder .NET zu tun haben. Verwenden Sie nicht »C#« im Suchbegriff. Das # verhindert eine sinnvolle Suche, weil der Suchserver dieses Zeichen ignoriert.

Eine andere hervorragende Adresse für die Recherche im Internet ist *groups.google.com/advanced_group_search*. In dieser speziellen Google-Suche können Sie nach aktuellen und archivierten Newsgroup-Beiträgen suchen. Die meisten Fragen, die ich beim Schreiben dieses Buchs hatte, konnte ich allein durch eine einfache Google-Recherche lösen.

In dem Feld NEWSGROUP der Google-Suche können Sie Ihre Suche auf eine oder mehrere Newsgroups einschränken (mehrere trennen Sie einfach durch Kommata oder über das Schlüsselwort `Or`). Wenn Sie den Namen der Newsgroup nicht kennen oder in mehreren Gruppen suchen wollen, können Sie den Namen mit Wildcards maskieren. Ich verwende eigentlich immer die Maske *dotnet*, weil alle .NET-Newsgroups diesen Begriff im Namen tragen.

Wenn Sie selbst in Newsgroups Fragen stellen oder Antworten geben wollen, empfehle ich die Newsgroup *microsoft.public.dotnet.csharp*. In dieser englischsprachigen Newsgroup wird sehr viel gepostet. In der deutschen Newsgroup *microsoft.public.de.german.entwickler.dotnet.csharp* finden Sie dagegen nur relativ selten neue Beiträge.

Im Internet finden Sie auch eine große Anzahl hervorragender Websites, die sich mit .NET oder C# auseinandersetzen. Ich liste diese hier aber nicht auf, weil Sie bei *www.geocities.com/csharplink* eine Seite mit vielen Links zu den besten (englischsprachigen) .NET- und C#-Websites finden. Deutschsprachige Websites tauchen dort nicht auf, deshalb empfehle ich Ihnen noch *www.sharpsite.de*.

1.4 Wichtige Begriffe

Bevor ich das .NET-Framework und die Common Language Runtime beschreibe, erläutere ich zunächst einige Begriffe, deren Kenntnis ich in den folgenden Abschnitten voraussetze.

1.4.1 Die OOP

C# ist eine rein objektorientierte Programmiersprache. Wenn Sie mit C# programmieren, arbeiten Sie permanent mit Klassen, Objekten und deren Methoden und Eigenschaften. Sie sollten also die Grundlagen der objektorientierten Programmierung (OOP) kennen. Diese

Grundlagen kann ich in diesem Buch aber nicht ausführlich klären. Lesen Sie dazu den Online-Artikel »Grundlagen der OOP«.

Die wichtigsten Begriffe, die für das Verständnis dieses Buchs bis Kapitel 4 (das die OOP mit C# behandelt) ausreichen, fasse ich aber kurz zusammen:

- **Klassen**: Eine Klasse enthält einzelne Teilprogramme einer objektorientierten Anwendung. OOP-Anwendungen bestehen aus mindestens einer, meist aber aus mehreren Klassen. Klassen enthalten die Deklaration von zusammengehörigen Methoden und Eigenschaften.
- **Methoden**: Eine Methode ist eine gespeicherte Folge von Programmanweisungen, die über den Namen der Methode aufgerufen werden kann. Methoden stellen somit einen einzelnen, elementaren Programmteil dar, der an verschiedenen Stellen innerhalb der Anwendung durch den Aufruf der Methode ausgeführt werden kann.
- **Eigenschaften**: Eine Eigenschaft dient dem Speichern von Daten innerhalb der Anwendung. Eigenschaften sind prinzipiell so etwas wie Variablen.
- **Objekte und Instanzen**: Um die in einer normalen Klasse enthaltenen Eigenschaften und Methoden verwenden zu können, müssen Sie eine Instanz dieser Klasse erzeugen. Eine Instanz einer Klasse wird auch als Objekt bezeichnet. Die Klasse dient dann quasi als Bauplan für das erzeugte Objekt. Das Objekt wird über eine Variable (oder eine andere Referenz) verwaltet. Über diese Variable können Sie die Methoden und Eigenschaften der Klasse verwenden. Dazu geben Sie den Namen der Objektvariable gefolgt von einem Punkt und dem Namen der Eigenschaft bzw. Methode an. Die Methoden und Eigenschaften, die nur über Instanzen von Klassen verwendet werden können, werden auch als **Instanzmethoden** bzw. **Instanzeigenschaften** bezeichnet.
- **Statische Klassenelemente**: Einige Klassen besitzen statische Methoden und Eigenschaften. Diese können Sie dann auch ohne Instanz der Klasse verwenden. Dazu müssen Sie lediglich den Klassennamen, getrennt durch einen Punkt vor den Namen der Methode oder Eigenschaft setzen. Statische Methoden werden

auch als **Klassenmethoden**, statische Eigenschaften als **Klasseneigenschaften** bezeichnet.

- **Klassenbibliotheken**: Klassenbibliotheken enthalten vorgefertigte Klassen in kompilierter Form. Klassenbibliotheken können Sie sehr einfach in Ihren Programmen einsetzen, um die Funktionalität der enthaltenen Klassen zu nutzen. Das .NET-Framework enthält eine Vielzahl an Klassenbibliotheken mit sehr vielen Klassen für die unterschiedlichsten Aufgaben.

1.4.2 Ereignisorientiertes Programmieren

Viele Objekte besitzen neben Eigenschaften und Methoden auch Ereignisse. Über Ereignisse gibt ein Objekt dem Programmierer die Möglichkeit, auf Anwendereingaben oder andere Ereignisse zu reagieren. Diese Art der Programmierung wird übrigens ereignisorientierte Programmierung genannt.

Im Gegensatz zu Konsolenanwendungen[2] laufen echte Windowsanwendungen nicht mehr ab einem definierten Einsprungpunkt sequenziell bis zum Ende ab, sondern warten beharrlich darauf, dass etwas passiert. Passiert nichts, warten diese Programme ewig (oder bis der Rechner ausgeschaltet wird). Wenn Sie mit modernen Programmierumgebungen eine Windowsanwendung schreiben, erzeugen Sie für alle Ereignisse, die Ihr Programm auswerten soll, Ereignisbehandlungsmethoden. Diese Methoden werden dann ausgeführt, wenn das Ereignis auf dem betreffenden Objekt eingetreten ist. Ist die Methode abgearbeitet, wartet Ihre Anwendung (oder um genauer zu sein: das aktive Fenster) auf weitere Ereignisse. Die Steuerelemente und Formulare mit denen Sie die Oberfläche einer Anwendung gestalten stellen Ihnen zu diesem Zweck meist gleich mehrere Ereignisse zur Verfügung. Eine Textbox (zur Ein- und Ausgabe von Texten) besitzt beispielsweise u. a. die Ereignisse `Enter` (der Anwender hat den Eingabecursor in die Textbox gesetzt), `Leave` (der Anwender hat den Eingabecursor aus der Textbox herausbewegt), `TextChanged` (der Anwender hat den Text geändert) und `DoubleClick` (der Anwender hat

2. Eine Konsolenanwendung verwendet die Konsole für Eingaben und Ausgaben (in Textform). Die Konsole ist vergleichbar mit der alten DOS-Eingabeaufforderung.

in der Textbox doppelt geklickt). Obwohl die meisten Steuerelemente und Formulare eine große Anzahl Ereignisse besitzen, reichen meist nur einige wenige zum Programmieren aus. Wenn Sie Visual Studio verwenden, hilft diese Entwicklungsumgebung bei der Erzeugung einer Ereignisbehandlungsmethode. Visual Studio beschreibe ich noch ausführlicher in Kapitel 2.

Die meisten Ereignisse bei Steuerelementen betreffen Anwendereingaben. Im Prinzip kann man sagen, dass alles, was ein Anwender (mit der Anwendung) anstellen kann, in einem Ereignis ausgewertet werden kann. So können Sie z. B. bei einem Textbox-Steuerelement darauf reagieren, dass der Anwender das Steuerelement verlassen will, indem Sie den eingegebenen Text auf Gültigkeit überprüfen. Am wichtigsten sind jedoch oft der OK- und der Schließen-Schalter auf einem Fenster, bei denen Sie im `Click`-Ereignis darauf reagieren, dass der Anwender den Schalter betätigt hat.

1.4.3 XML

Wenn unter .NET Daten in einfachen Dateien gespeichert oder über ein Netzwerk versendet werden sollen, verwenden die entsprechenden .NET-Tools dazu eigentlich immer XML. Bereits im nächsten Kapitel, bei der Beschreibung von Visual Studio, erscheint der Begriff »XML«, weil Visual Studio verschiedene Projekt- und Konfigurationsinformationen in solchen Dateien speichert. Deshalb sollten Sie wissen, was XML grundsätzlich ist.

Ich beschreibe XML hier nur grundlegend, so, wie XML innerhalb von .NET üblicherweise verwendet wird. Weitere Informationen finden Sie im XML-Buch der Nitty-Gritty-Reihe oder im Internet. Bei members.aol.com/xmldoku finden Sie einen deutschsprachigen Artikel zu XML, bei www.w3.org/TR/REC-xml die offizielle XML-Dokumentation.

Was ist XML?

XML (Extensible Markup Language) ist eine Familie von Techniken zur Speicherung und Verarbeitung von strukturierten Daten in Textform. XML-Dateien können Sie sich als Datenbanken vorstellen (die ja auch

strukturierte Daten speichern). XML besitzt aber einige Vorteile gegenüber Datenbanken: Die Daten werden in Textform gespeichert, die Struktur der Daten ist absolut flexibel (in Datenbanken ist die Struktur festgelegt), verschiedene XML-Technologien erleichtern die Arbeit mit den gespeicherten Daten. Jedes Programm, das XML-fähig ist, kann diese Daten auslesen und weiterverarbeiten. Das ist der große Vorteil von XML.

Der XML-Standard besteht nicht nur aus der Festlegung, wie Daten gespeichert werden sollen, sondern definiert auch andere Techniken. XSL-Stylesheets werden z. B. eingesetzt, um das Layout von XML-Daten zu beschreiben. XML-Schemata werden verwendet, um die Struktur eines XML-Dokuments zu beschreiben (damit ein XML-fähiges Programm diese Struktur erkennen kann). Diese Techniken beschreibe ich hier allerdings nicht. Für dieses Buch reicht es aus, wenn Sie wissen, wie eine einfache XML-Datei aufgebaut ist.

Die Grundstruktur einer XML-Datei

XML-Dateien bestehen aus einzelnen Elementen, die über Tags definiert werden. Ein Tag besitzt einen Namen und wird in spitze Klammern eingeschlossen. Ein komplettes XML-Element wird mit einem Start-Tag eingeleitet und durch ein Ende-Tag beendet. Das Ende-Tag sieht prinzipiell aus wie das Start-Tag, nur dass dem Tagnamen ein Schrägstrich vorangestellt wird. Der Inhalt des Elements stellt die gespeicherten Daten dar:

```
<Elementname>Inhalt</Elementname>
```

Das folgende Element speichert z. B. einen Buchtitel:

```
<titel>C#</titel>
```

Ein XML-Element kann nicht nur einen Inhalt besitzen, sondern auch Attribute, die zusätzliche Informationen zum Element liefern. Attribute werden im Starttag definiert:

```
<Elementname Attributname="Wert"[...]>
 Inhalt</Elementname>
```

Der Wert eines Attributs wird immer in Anführungszeichen eingeschlossen. Einzelne Attribute werden einfach durch Leerzeichen voneinander getrennt (das `[...]` in der Syntaxbeschreibung deutet an,

dass Sie mehrere Attribute im Starttag unterbringen können). Soll zum Beispiel das Gewicht eines Buchs gespeichert werden, kann ein Attribut die Einheit definieren:

```
<gewicht einheit="g">400</gewicht>
```

XML wird immer unter Beachtung der Groß- und Kleinschreibung ausgewertet. Achten Sie also darauf, dass Sie die Namen der Elemente und der Attribute in der korrekten Schreibweise angeben, wenn Sie XML-Dokumente bearbeiten.

Für XML-Elemente ohne Inhalt kann auch eine Kurzform verwendet werden, bei der das Start- und das Ende-Tag zusammengefasst sind:

```
<Elementname/>
```

Sinnvoll ist diese Kurzform für Elemente, die lediglich über Attribute definiert sind.

XML-Elemente können geschachtelt werden. Untergeordnete Elemente gehören dann zum übergeordneten Element. Ein XML-Dokument kann damit mehrere »Datensätze« speichern und baut sich auf wie ein Baum. So können Sie z. B. die Daten mehrerer Buchtitel in einem XML-Dokument speichern.

Der Standard verlangt, dass ein XML-Dokument in der ersten Zeile in einer speziellen Anweisung seinen Typ und die verwendete XML-Version deklariert. Optional können Sie in dieser Zeile noch weitere Attribute des Dokuments, wie z. B. den zu verwendenden Zeichensatz definieren. Die Angabe des Zeichensatzes ist eigentlich immer notwendig, damit das lesende Programm die gespeicherten Zeichen korrekt auswertet. In unseren Breitengraden wird normalerweise der Zeichensatz ISO-8859-1 verwendet (auch bekannt als »Western Latin 1«). Zusätzlich dazu verlangt der Standard, dass ein XML-Dokument immer nur ein einziges Wurzel-Element (Root) enthält. Ein XML-Dokument, das Buchtitel speichert, sieht dann z. B. so aus wie im folgenden Listing:

```
<?xml version="1.0" encoding="ISO-8859-1"
 standalone="yes" ?>
<buecher>

 <buch isbn="3-8273-1671-5">
   <reihe>Nitty-Gritty</reihe>
   <titel>Java</titel>
   <gewicht einheit="g">390</gewicht>
   <autor>
     <vorname>Florian</vorname>
     <nachname>Hawlitzek</nachname>
   </autor>
 </buch>

 <buch isbn="3-8273-1856-4">
   <reihe>Nitty-Gritty</reihe>
   <titel>C#</titel>
   <gewicht einheit="g">400</gewicht>
   <autor>
     <vorname>Jürgen</vorname>
     <nachname>Bayer</nachname>
   </autor>
 </buch>

</buecher>
```

Das `standalone`-Attribut der XML-Deklaration sagt aus, dass dieses Dokument nicht von anderen Dateien abhängig ist. Die Datei speichert die Daten von zwei Büchern. Für jedes Buch ist im Attribut *isbn* die ISBN-Nummer gespeichert, damit das Buch eindeutig identifizierbar ist. Die ISBN-Nummer könnte alternativ auch in einem eigenen Element gespeichert werden, das dem *buch*-Element untergeordnet ist. Das Element *autor* besitzt zwei untergeordnete Elemente, die die Daten des Autors speichern.

1.5 C# und das .NET-Framework

C# ist eine C++-ähnliche, neue Programmiersprache, die die Flexibilität von C++ mit den RAD[3]-Features von Visual Basic vereinigt. C# ist aber lange nicht so kompliziert wie C++ und ersetzt gefährliche Sprachmerkmale wie Zeiger und den direkten Speicherzugriff durch wesentlich mächtigere Techniken (lässt Ihnen aber auch den Freiraum, die »alten« Techniken doch noch zu verwenden). In vielen Bereichen von C# finden Sie auch viel Ähnlichkeit mit Java. C# basiert, wie alle .NET-Sprachen, auf dem .NET-Framework, das – vereinfacht gesagt – eine Laufzeitumgebung für Programme zur Verfügung stellt.

Die von C# verwendete Klassenbibliothek ist kein fester Bestandteil der Sprache, sondern steht gleich mehreren (.NET-)Programmiersprachen (wie z. B. auch Visual Basic.NET) zur Verfügung. Aufgrund der Mächtigkeit dieser Bibliothek finden Sie sehr viele fertige Problemlösungen darin.

1.5.1 .NET

Dieses Buch ist kein Buch über .NET, sondern behandelt die Programmierung unter .NET. Ich beschreibe die .NET-Idee hier also nur kurz. Im Internet finden Sie auf sehr vielen Seiten grundlegende Informationen zu .NET.

.NET (gesprochen häufig als »Dotnet«) ist eine neue Strategie von Microsoft zur Erzeugung und Verteilung von Software. Die Grundlage von .NET ist das .NET-Framework. Das .NET-Framework ist eine komplexe (aber einfach anzuwendende) Infrastruktur, in der Sie Anwendungen programmieren, kompilieren, ausführen und verteilen können. .NET unterstützt die Programmierung von Windows- und Internetanwendungen. .NET-Anwendungen können auf einfache Weise über das Internet (oder über andere Netze) mit anderen Anwendungen oder Komponenten kommunizieren, die auf entfernten Rechnern ausgeführt werden. Das dazu verwendete Protokoll, SOAP, basiert auf HTTP und XML und kann deswegen von den verschiedensten Systemen verwendet werden. Mit ASP.NET können Sie auf einfa-

3. Rapid Application Development = Schnelle Anwendungsentwicklung

che Weise Anwendungen entwickeln, die auf einem Webserver ausgeführt werden und die HTML-Code erzeugen, der in einem Browser dargestellt werden kann. Webdienste sind Dienste, die auf einem Webserver ausgeführt werden und die von einer auf einem entfernten Rechner laufenden, .NET-fähigen Anwendung verwendet werden können. Diese Dienste sind im Prinzip Funktionen, die über das Internet aufgerufen werden können. Ein Webdienst, der auf dem Webserver einer Bank läuft, kann z. B. den aktuellen Kurs von Aktien über das Internet verfügbar machen. .NET stellt Ihnen damit alle Möglichkeiten zur Verfügung, die Sie für die Softwareentwicklung benötigen.

Neben dem für Programmierer wichtigen Framework steht .NET aber auch für eine Vision. Basierend auf den neuen Webdiensten und den damit verbundenen Technologien verfolgt Microsoft mit .NET eine Idee, die als »Software as a Service« bezeichnet wird. Anwender sollen in Zukunft ihre Programme nicht mehr lokal auf dem Rechner speichern, sondern mehr oder weniger aus einzelnen Webdiensten und/oder den Funktionen von (Microsoft-)Produkten zusammenstellen. Die neuen Versionen der einzelnen Microsoft-Produkte werden deshalb durchgehend mit .NET-Support ausgestattet.

1.5.2 Das .NET-Framework

Das .NET-Framework ist der grundlegende Teil von .NET. Dieses Framework stellt Anwendungen (normalerweise) alle benötigten Dienste zur Verfügung. Das unterscheidet die .NET-Programmierung erheblich von der klassischen Windowsprogrammierung, bei der Programmierer häufig auf vollkommen unterschiedliche Dienste, wie das Windows-API oder COM-Komponenten, zugreifen mussten.

Das .NET-Framework besteht hauptsächlich aus:

- einer umfangreichen Klassenbibliothek, die Klassen für alle grundlegenden Problemlösungen enthält,
- der CLR (Common Language Runtime), eines Dienstes, der die grundlegenden Technologien zur Verfügung stellt und .NET-Programme ausführt und
- verschiedenen Compilern für unterschiedliche Programmiersprachen.

Abbildung 1.1 zeigt eine Übersicht über das .NET-Framework.

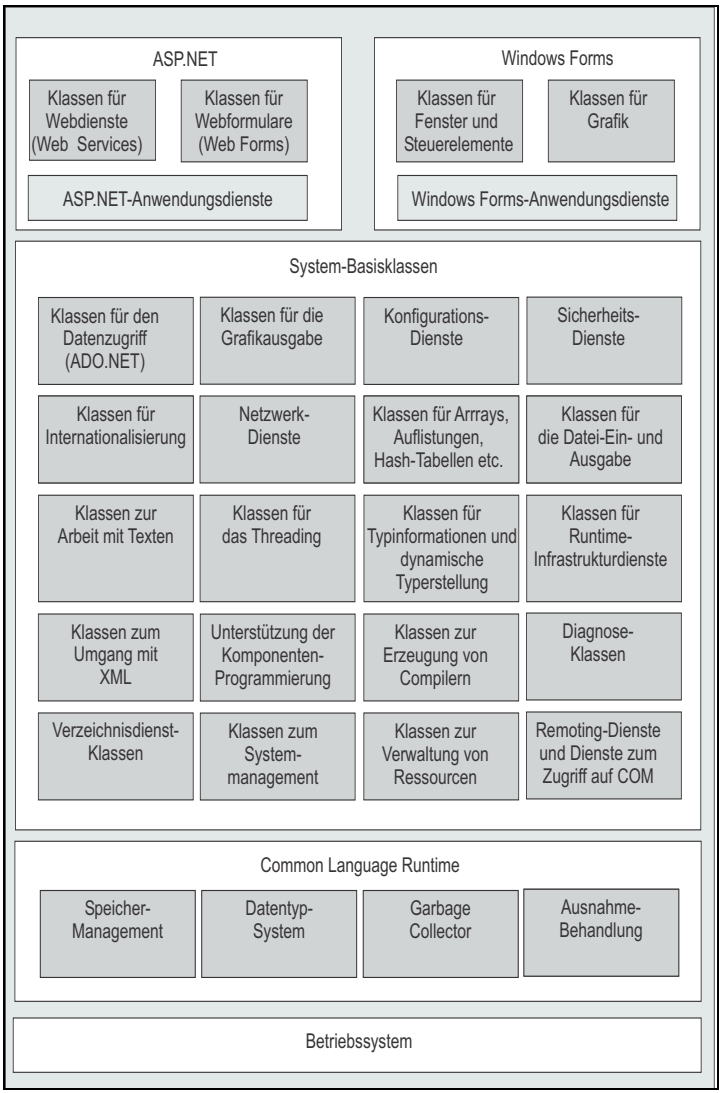

Bild 1.1: Das .NET-Framework (ohne Compiler)

Das Betriebssystem liefert natürlich die grundlegenden Dienste für alle Programme (bei Windows in Form des Windows-API). Diese Systemdienste werden allerdings von der Common Language Runtime gekapselt.

Der IL-Code und der Just-In-Time-Compiler

.NET-Programme basieren normalerweise nicht mehr direkt auf dem Windows-API, sondern bestehen aus einem speziellen Microsoft-Zwischencode, der auch als Intermediate Language (IL) oder Managed Code bezeichnet wird. Verschiedene sprachspezifische Compiler erzeugen aus dem Quellcode eines Programms IL-Code. Microsoft stellt Compiler für C++, C#, Visual Basic und JScript zur Verfügung. Viele externe Hersteller bieten Compiler für die verschiedensten Sprachen, wie z. B. Eiffel, an. Welche Sprache Sie zur Erzeugung von .NET-Anwendungen verwenden, ist eigentlich nur noch Geschmackssache.

Der vom Compiler erzeugte Zwischencode kann nicht direkt vom Betriebssystem ausgeführt werden, weil das Betriebssystem mit den speziellen Befehlen gar nichts anfangen kann. Um den Code ausführbar zu machen, wird dieser beim Aufruf von einem Just-In-Time-Compiler (JIT) in Betriebssystemcode umgewandelt. Der Just-In-Time Compiler kompiliert dabei immer nur die Funktion, die gerade angesprochen wurde, behält den kompilierten Code für den nächsten Aufruf dann aber in seinem Cache. Daraus ergeben sich einige Vorteile:

- IL-Programme können auch auf anderen Betriebssystemen ausgeführt werden, sofern das .NET-Framework auf dem System verfügbar ist,
- die CLR, die die Programme ausführt, kann diese daraufhin überprüfen, ob die Ausführung sicher ist (d. h., nicht zu Abstürzen führen kann),
- der JIT kann den IL-Code für den jeweiligen Prozessor optimieren.

Die prozessoroptimierte Just-In-Time-Kompilierung bewirkt schon, dass .NET-Anwendungen sehr performant sind. Um diese zusätzlich zu optimieren, können Sie den IL-Code bei der Installation oder nachträglich über ein spezielles Tool des .NET-Framework, den »Native Code Generator« (*ngen.exe*), auch dauerhaft in nativen Maschinencode umwandeln.

Die Common Language Runtime

Die Common Language Runtime (CLR) ist eine Laufzeitumgebung für alle .NET-Programmiersprachen. Sie

- führt den IL-Code aus, der von einer beliebigen .NET-Programmiersprache erzeugt wurde. Vor der Ausführung wird der IL-Code daraufhin überprüft, ob er sicher ist (d. h. keine Speicherbereiche überschreibt, die nicht zum aktuell ausgeführten Programmteil gehören). Abstürze, die durch ein versehentliches Überschreiben von fremden Speicherbereichen entstehen, sind unter der CLR nicht mehr möglich, weil diese unsicheren Programmcode erst gar nicht ausführt. Sie erzeugt stattdessen eine Ausnahme (Exception), die im Programm abgefangen werden kann,
- bietet den Programmen eine große Anzahl an Standarddatentypen, die den Anforderungen moderner Programmiersprachen genügen,
- definiert mit dem Common Type System (CTS) die Regeln, die festlegen, wie neue Datentypen (Strukturen, Klassen etc.) konstruiert werden müssen. Diese Regeln müssen von allen .NET-Programmiersprachen eingehalten werden, wenn diese CTS-kompatible Programme erzeugen sollen,
- übernimmt das Management von Objekten und deren automatische Zerstörung (über einen Garbage Collector, siehe Seite 33),
- übernimmt den Aufruf von Methoden in Objekten,
- und das Handling von Ausnahmen (Exceptions).

Keine .NET-Anwendung sollte die Dienste des Betriebssystems direkt verwenden (was aber beispielsweise mit C++ auch möglich ist). Der Zugriff sollte immer (mindestens) über die CLR erfolgen. Weil diese wichtigen Basisfeatures von allen .NET-Programmiersprachen verwendet werden, ist es mit .NET möglich, dass ein in einer Programmiersprache geschriebenes Programm (bzw. eine Komponente) ohne Probleme von einem in einer anderen Programmiersprache geschriebenen Programm verwendet wird. Die in der alten Welt bestehenden Probleme der Zusammenarbeit von Programmen, die in unterschiedlichen Sprachen geschrieben wurden (inkompatible Datentypen, unterschiedliches Handling von Exceptions etc.) gehören damit der Vergangenheit an.

Wenn Sie zum Beispiel in C# eine Komponente entwickeln, diese kompilieren und an andere Programmierer weitergeben, können diese die Klassen Ihrer Komponente z. B. in Visual Basic.NET ohne Probleme verwenden. Die Technik zur Verwendung von Objekten und zum Aufruf von Methoden und die verwendeten Datentypen sind eben identisch. Der VB-Programmierer kann sogar von Ihren Klassen neue Klassen ableiten (sofern Sie das nicht unterbinden). Löst Ihre Komponente bei bestimmten Ausnahmezuständen (oder im Fehlerfall) eine Ausnahme aus, kann der VB-Programmierer diese ganz einfach in einem Ausnahmeblock abfangen.

Der Garbage Collector

Alle .NET-Programme, die auf der CLR aufsetzen, besitzen einen Thread[4], der als Garbage Collector (»Müllsammler«) bezeichnet wird. Immer dann, wenn die Anwendung gerade nicht besonders beschäftigt ist, geht der Garbage Collector durch den Speicher und entfernt Objekte, die nicht mehr verwendet werden. Ein .NET-Programmierer muss sich also nicht um die Speicherbereinigung kümmern. Programmfehler und Speicherlöcher, wie diese in Sprachen wie C++ oder Delphi möglich sind (die eine explizite Freigabe der Objekte erfordern), sind damit ausgeschlossen. Die Eigenheit des Garbage Collectors, Objekte nicht sofort, sondern erst zeitverzögert zu zerstören, muss bei der Programmierung ein wenig im Auge behalten werden. In Kapitel 4 gehe ich noch darauf ein.

1.5.3 Die Möglichkeiten von .NET-Programmen

Mit einer .NET-Programmiersprache können Sie alle modernen Softwarearchitekturen realisieren.

Einfache Windowsanwendungen

Einfache Windowsanwendungen erstellen Sie mit Hilfe der Klassen der `Windows.Forms`-Bibliothek. Dazu stehen Ihnen Formulare (Windows-Fenster) und eine umfangreiche Anzahl von Steuerelementen zur Verfügung.

4. Ein Thread (Programmfaden) ist ein Programmteil, der parallel und quasi gleichzeitig zu anderen Programmteilen ausgeführt wird.

Komponentenbasierte Anwendungen

Komponentenbasierte Anwendungen sind nicht in einer einzelnen ausführbaren Datei gespeichert, sondern nutzen daneben eine oder mehrere Komponenten. Der Begriff »Komponente« ist ein eher allgemeiner Begriff. Unter C# sind mit Komponenten eigentlich Klassenbibliotheken gemeint. Über Klassenbibliotheken können Sie Programmteile so auslagern, dass Sie diese in anderen Projekten bzw. Programmen ganz einfach wiederverwenden können. Die Programmierung und Anwendung von Klassenbibliotheken ist unter C# (und unter .NET im Allgemeinen) sehr einfach. Ich beschreibe Klassenbibliotheken in Kapitel 4.

Client/Server-Anwendungen mit SOAP und XML

Klassenbibliotheken können auch auf zentralen Rechner gespeichert und über das so genannte Remoting, unter Verwendung des SOAP[5]-Protokolls von entfernten Rechnern aus verwendet werden. Eine zentrale Klassenbibliothek kann dann gleich mehrere Anwendungen auf unterschiedlichen Rechnern bedienen. SOAP ist ein einfaches, textbasiertes Protokoll für den Zugriff auf Objekte, die in entfernten Komponenten verwaltet werden. SOAP verwendet für den Transport von Daten zwischen einer Anwendung und einer Komponente XML. Da SOAP auf dem HTTP-Protokoll basiert, können Anwendungen die Dienste von Komponenten auch über das Internet verwenden. Die Entwicklung von Client/Server-Anwendungen ist damit sehr einfach.

Das detaillierte Verständnis von SOAP und XML ist für die Programmierung mit C# nicht unbedingt erforderlich. Sie verwenden diese Technologien einfach im Hintergrund. Wenn Sie trotzdem mehr wissen wollen, gehören anderen finden Sie sehr gute Informationen über SOAP auf der Internetseite msdn.microsoft.com/xml/general/soapspec.asp und über XML auf der Seite msdn.microsoft.com/xml/general.

5. Simple Object Access Protocol

Internetanwendungen

Mit SOAP ist es zunächst einmal recht einfach, Anwendungen zu schreiben, die entfernte Komponenten verwenden. Diese Komponenten können auch auf einem Computer installiert sein, der irgendwo im Internet läuft. Diese Internetanwendungen sind dann normalerweise echte Windowsanwendungen, die auf dem Client-Rechner installiert werden müssen.

Webdienste sind im Prinzip auch Komponenten, die auf entfernten Rechnern laufen. Webdienste werden aber über einen Webserver ausgeführt und können über die Internetadresse dieses Servers angesprochen werden. Webdienste vereinfachen die Verwendung von Komponenten über das Internet. Diese Dienste werden wohl auch hautsächlich von Windowsanwendungen genutzt.

Das Internet bietet aber auch die Möglichkeit, den Webbrowser als Oberfläche für eine Anwendung zu verwenden. ASP.NET ermöglicht Ihnen auf einfache Weise, den größten Teil einer browserbasierten Internetanwendung in Form von Webformularen auf dem Webserver auszuführen.

1.5.4 Typen, Namensräume, Assemblierungen und Module

Bei der Arbeit mit .NET begegnen Ihnen immer wieder die Begriffe »Typ«, »Namensraum« (Namespace), »Assemblierung« (Assembly) und »Modul«. Damit Sie wissen, worum es sich dabei handelt, beschreibe ich diese Begriffe kurz.

Typ

Jeder Datentyp wird in .NET als Typ (Type) bezeichnet. Dazu gehören Klassen, Strukturen, Aufzählungen (Enumerations), Schnittstellen und Arrays. Die Standarddatentypen wie `Integer`, `Double` und `String` sind in .NET übrigens grundsätzlich Klassen.

Namensräume

Das .NET-Framework fasst Typen immer in so genannten Namensräumen (Namespaces) zusammen. Ein Namensraum ist eine logische Gruppierung von Typen. Der Namensraum `System.IO` enthält z. B. alle Typen, die für die Arbeit mit dem Dateisystem verwendet werden. Die

Gruppierung vereinfacht die Suche nach Typen, erleichtert deren Zuordnung und verhindert Kollisionen von gleichnamigen Typen. Über Namensräume ist es meist recht einfach, in der Klassenbibliothek des .NET-Framework einen Typ zu finden, der eine bestimmte Funktionalität bietet. Sie müssen dazu prinzipiell nur beginnend beim `System`-Namensraum die einzelnen Namensräume durchgehen.

Namensräume werden normalerweise geschachtelt. Der `System`-Namensraum des .NET-Framework enthält z. B. einige Typen, aber auch wieder Namensräume. In `System` finden Sie z. B. die Namensräume `Data` (Klassen zur Arbeit mit Daten) und `Drawing` (Klassen zum Zeichnen von grafischen Elementen). Der `Data`-Namensraum enthält wiederum (u. a.) die Namensräume `OleDb` (die Klassen der zum Zugriff auf Datenbanken verwendeten ActiveX Data Objects, deren Basistechnologie OLEDB ist) und `SqlClient` (Klassen zum direkten Zugriff auf den SQL-Server).

Die Dokumentation zum .NET-Framework stellt diese Namensräume sehr übersichtlich dar (Abbildung 1.2).

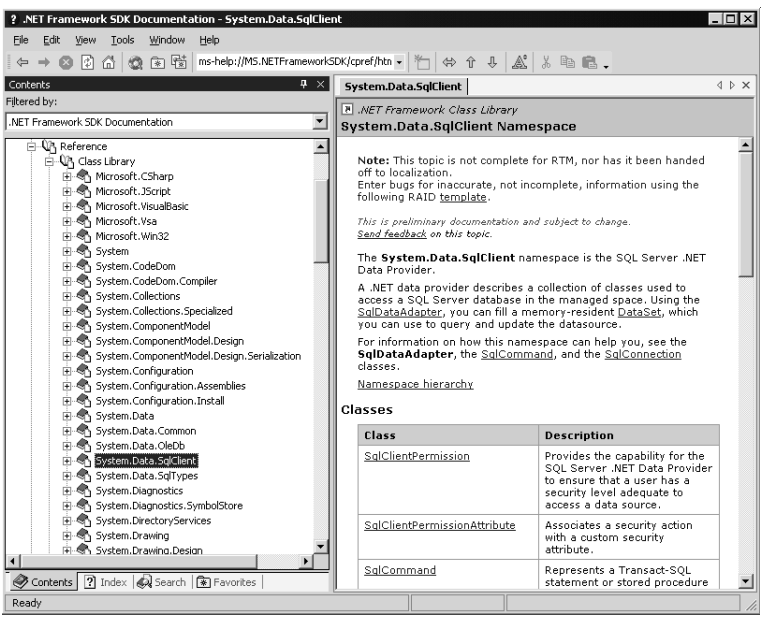

Bild 1.2: Die .NET-Framework-Dokumentation

Wenn Sie selbst Klassen entwickeln, können Sie diese in eigenen Namensräumen anlegen und diese ebenfalls schachteln.

Assemblierungen und Module

Eine Assemblierung (englisch: »Assembly«) ist die Basiseinheit für die Verwaltung von kompiliertem Programmcode und für die Verteilung von Anwendungen. Eine Assemblierung enthält den kompilierten Code einer oder mehrerer Klassen und anderer Typen (Auflistungen, Strukturen etc.), alle Ressourcen (Bitmaps, Icons etc.), die von diesen Typen verwendet werden, und Metadaten, die die Assemblierung beschreiben.

Eine Anwendung ist eine Kombination mehrerer Assemblierungen. Auch wenn Sie in Ihren Anwendungen selbst keine Assemblierungen einsetzen, verwenden Sie zumindest die Assemblierungen des .NET-Frameworks. Normalerweise enthalten einzelne Assemblierungen Typen und Ressourcen, die thematisch zueinander passen. Die Assemblierung *System.IO.dll* enthält z. B. Klassen und andere Typen für die Eingabe und Ausgabe von Daten.

Eine Assemblierung kann aus einer einzelnen kompilierten Datei, aber auch aus mehreren Dateien bestehen. Die einzelnen Dateien werden als Module bezeichnet. Ein Modul enthält kompilierten Zwischencode (IL-Code), optional Ressourcen (z. B. Bitmaps und Icons) und Metadaten in Form einer .DLL- oder .EXE-Datei.

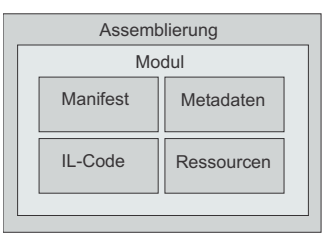

Bild 1.3: Eine Assemblierung, die nur aus einem Modul besteht

Metadaten beschreiben die im Modul enthaltenen Typen (also z. B. welche Klassen enthalten sind und wie die Eigenschaften, Methoden und Ereignisse der Klassen deklariert sind). Ein Modul der Assemblierung verwaltet normalerweise das so genannte Manifest (das aber

auch in einer separaten Datei gespeichert sein kann). Das Manifest verwaltet die Abhängigkeiten der Assemblierung (inkl. der Version der verknüpften Ressourcen), verweist auf die enthaltenen Module, speichert den Namen des Autors, definiert eine Versionsnummer und beschreibt, welche (System-)Zugriffsrechte zur Ausführung der Assemblierung notwendig sind.

Damit beschreibt eine Assemblierung sich vollständig selbst. Es sind also keine weiteren Informationen notwendig, wie es z. B. im veralteten COM-Modell in Form von Registry-Eintragungen notwendig war. Deshalb kann eine Assemblierung einfach auf einen Computer kopiert und dort verwendet werden, was die Verteilung von .NET-Anwendungen im Vergleich zum COM-Modell erheblich vereinfacht.

Assemblierungen im Anwendungsordner und im Assemblierungen-Cache

Assemblierungen können einfach in dem Ordner gespeichert werden, in dem die Anwendung abgelegt ist, die diese Assemblierung verwendet. Für selbst entwickelte Assemblierungen, die nur von einer Anwendung verwendet werden, wird das die Regel sein. Jeder Computer auf dem das .NET-Framework installiert ist, besitzt aber auch einen Ordner für Assemblierungen, der als Assemblierungen-Cache (Assembly Cache) bezeichnet wird. Diesen Cache finden Sie im Windows-Ordner unter dem Namen ASSEMBLY. Er speichert u. a. globale Assemblierungen, die von mehreren verschiedenen Anwendungen verwendet werden und solche, die vor dem versehentlichen Löschen durch den Benutzer geschützt werden sollen. Einige Assemblierungen des .NET-Framework sind Bestandteil des globalen Teils des Assemblierungen-Cache.

Versionsverwaltung

Das Manifest einer Assemblierung enthält die Versionsnummer (in der Form *Major.Minor.Build.Revision*) der Assemblierung selbst und verwaltet die Versionsnummern aller referenzierten globalen Assemblierungen. Globale (gemeinsame) Assemblierungen, die im Assemblierungen-Cache gespeichert werden, können ohne Probleme auch in mehreren Versionen auf einem System gespeichert sein. Die CLR verwendet die Versionsnummer, um die korrekte Version einer refe-

renzierten, globalen Assemblierung zu laden. Dies stellt sicher, dass ein Programm immer die Version der (globalen) Assemblierung verwendet, mit der es kompiliert wurde. Damit werden Versionsprobleme vermieden.

Bei lokalen Assemblierungen, die grundsätzlich immer im Anwendungsordner gespeichert sind (bzw. sein sollten), verwendet die CLR grundsätzlich immer die Version der Assemblierung, die im Anwendungsordner gespeichert ist. Ist dort versehentlich eine ältere oder neuere Version gespeichert (was eigentlich nicht vorkommen sollte), tritt eine Ausnahme (ein Laufzeitfehler) auf, wenn das Programm eine nicht (mehr) vorhandene Klasse, Methode oder Eigenschaft oder eine Methode oder Eigenschaft mit einer geänderten Signatur[6] verwendet. Die Versionsverwaltung funktioniert also nur für globale Assemblierungen. Da es wahrscheinlich nie vorkommt, dass die Version einer lokalen Assemblierung mit einer älteren oder neueren Version überschrieben wird, ist das nicht weiter schlimm.

Verwaltung der erforderlichen Zugriffsrechte

Im Manifest ist eine Liste aller Zugriffsrechte gespeichert, die von der Assemblierung gefordert werden. Daneben existiert eine Liste von Rechten, die erwünscht sind und eine Liste solcher Rechte, die explizit abgelehnt werden. Eine Assemblierung, die das Senden, Empfangen, Editieren und Anzeigen von E-Mails ermöglicht, benötigt z. B. die Rechte, über das Netzwerk auf den Ports 110 (POP3) und 25 (SMTP) zu kommunizieren, sie erwünscht das Recht, JavaScript-Programme auszuführen, um die volle Funktionalität von E-Mails im HTML-Format zu ermöglichen und lehnt Rechte ab, auf den Datenträger zu schreiben oder das lokale Addressbuch zu lesen, um E-Mail-Viren keinen Spielraum zu lassen[7].

6. Die Signatur einer Methode setzt sich zusammen aus den Datentypen, der Reihenfolge, der Anzahl der Argumente und dem Rückgabewert.
7. O.K., dieses Beispiel habe ich dem Buch »C# Essentials« entnommen, aber es ist eben ein gutes Beispiel.

Assemblierungen und Namensräume

Normalerweise enthält eine Assemblierung alle Klassen eines (untergeordneten) Namensraums. Die Klassen des Namensraums `System.Net` sind z. B. in der Datei *system.net.dll* gespeichert. Wenn Sie in Ihrem Quellcode mit den Klassen einer Assemblierung arbeiten wollen, müssen Sie auf diese verweisen. Innerhalb von Visual Studio geht das ganz einfach über das Kontextmenü des VERWEISE-Ordners im Projektmappen-Explorer. Beim Kompilieren über den Kommandozeilen-Compiler müssen Sie die Assemblierung über spezielle Befehlszeilen-Argumente angeben. Wie Sie das machen, beschreibe ich in Kapitel 2.

2 Einführung in C#

Bevor ich in Kapitel 3 die Sprachelemente von C# beschreibe, zeige ich in diesem Kapitel, wie Sie C#-Programme prinzipiell entwerfen und ausführen. Dazu gehört zunächst, dass Sie wissen, welche Möglichkeiten Sie mit C# haben. Danach zeige ich, wie der Quellcode eines C#-Programmms grundlegend aufgebaut wird. Schließlich beschreibe ich noch, wie Sie C#-Programme in Visual Studio erzeugen und wie Sie alternativ dazu den C#-Compiler direkt verwenden.

2.1 Die C#-Projekttypen

Mit C# können Sie einige verschiedene Arten von Anwendungen erzeugen. Dieses Buch beschäftigt sich hauptsächlich mit Windowsanwendungen und Klassenbibliotheken. Meine Internetprogrammierungsbücher in der Nitty-Gritty- und der Goto-Reihe setzen sich u. a. mit Internetanwendungen auseinander.

Tabelle 2.1 zeigt die Möglichkeiten, die Sie mit C# besitzen.

Anwendung	Beschreibung
Windowsanwendung	Eine Windowsanwendung nutzt Windows-Fenster (Formulare) und -Steuerelemente für die Benutzeroberfläche. Über Ereignisse der Formulare und Steuerelemente können Sie auf Benutzereingaben (oder andere Ereignisse) reagieren.
Klassenbibliothek	Ein Klassenbibliothek-Projekt erzeugt eine Assemblierung, die eine oder mehrere Klassen enthält. Diese Klassen können Sie dann in anderen Anwendungen verwenden, indem Sie die Assemblierung referenzieren.
Windows-Steuerelementbibliothek	Ein Windows-Steuerelementbibliothek-Projekt erzeugt eine Assemblierung, die selbst entwickelte Steuerelemente enthält, die Sie in Windows-Anwendungen verwenden können.

Anwendung	Beschreibung
ASP.NET-Webanwendung	Eine ASP.NET-Webanwendung enthält Webformulare mit speziellen Steuerelementen, die prinzipiell programmiert werden, wie bei Windows-Anwendungen (mit Ereignissen). Im Unterschied zu einer Windows-Anwendung wird eine Webanwendung von einem Webserver ausgeführt und erzeugt HTML-Code für die Benutzerschnittstelle.
ASP.NET-Webdienst	Wie ich es bereits in Kapitel 1 beschrieben habe, ist ein Webdienst eine Komponente, die auf einem Webserver ausgeführt wird und deren Methoden über das Internet aufgerufen werden können.
Web-Steuerelementbibliothek	Ein solches Projekt erzeugt eine Assemblierung, die selbst entwickelte Steuerelemente enthält, die Sie in Webanwendungen verwenden können.
Konsolenanwendung	Eine Konsolenanwendung läuft unter Windows direkt an der Konsole (dem »DOS-Prompt«). Ein- und Ausgaben erfolgen bei einer solchen Anwendung in Textform an der Konsole.
Windows-Dienst	Ein Windows-Dienst läuft unter Windows im Hintergrund. Im Prinzip ist ein Windows-Dienst ein Programm ohne Oberfläche, das automatisch ausgeführt wird und das bestimmte Dienste anbietet, die von anderen Anwendungen verwendet werden können. Der SQL Server und der Internet Information Server laufen z. B. als Windows-Dienst.

Tabelle 2.1: Die verschiedenen Anwendungen, die mit C# erzeugt werden können

2.2 Kompilieren ohne Visual Studio

Obwohl Visual Studio Ihnen eine große Anzahl an Hilfsmitteln bietet, können Sie C#-Programme auch ohne diese Entwicklungsumgebung

erzeugen und kompilieren. Im einfachsten Fall schreiben Sie die Anwendung in einem einfachen Editor und kompilieren diese über den Kommandozeilencompiler.

Der Kommandozeilencompiler

Die ersten Beispiele dieses Buchs sind einfache Anwendungen, die ohne Probleme in einem einfachen Editor entwickelt werden können. Damit Sie diese Quellcodes auch ohne Visual Studio ausprobieren können, zeige ich zunächst, wie Sie den Kommandozeilen-Compiler verwenden. Dieser ist unter dem Namen *csc.exe* im Ordner *\WINNT\Microsoft.NET\Framework\vx.x.xxxx* gespeichert, wobei *x.x.xxxx* für die aktuelle Versionsnummer des .NET-Framework steht.

Um beim Aufruf von *csc.exe* nicht immer den Ordner mit angeben zu müssen, sollten Sie diesen in den Windows-Pfad aufnehmen. Unter Windows 2000 finden Sie den dazu notwendigen Dialog in der Systemsteuerung unter dem Eintrag SYSTEM. Klicken Sie dort auf das Register ERWEITERT und dann auf UMGEBUNGSVARIABLEN. Suchen Sie in der unteren Liste die Variable PATH und klicken auf BEARBEITEN. Fügen Sie dem Pfad das .NET-Framework-Verzeichnis hinzu, indem Sie das Verzeichnis hinten, durch ein Semikolon getrennt, anhängen.

Nun können Sie den Kommandozeilen-Compiler (und andere .NET-Tools) in jedem Ordner des Systems direkt aufrufen. Eine Quellcodedatei kompilieren Sie einfach über

```
csc [Optionen] [Dateiname(n)]
```

Die Optionen des Compilers sind sehr vielfältig. Sie können diese abfragen, indem Sie `csc /?` eingeben. Die wichtigsten zeigt Tabelle 2.2.

Option	Bedeutung
`/out:`*Dateiname*	In dieser Option können Sie den Dateinamen der zu erzeugenden Assemblierung angeben. Falls Sie diese Option nicht verwenden, erhält die Assemblierung den Namen der ersten Quellcodedatei mit einer zum Typ der Assemblierung passenden Endung.

Option	Bedeutung
/recurse: *Platzhalter*	Wenn Sie alle Quellcodedateien eines Ordners in die Assemblierung kompilieren wollen, können Sie diese einfach über diese Option einbinden. /recurse:*.cs kompiliert z. B. alle Dateien mit der Endung *.cs*.
/target: {exe | winexe | module | library}	Standardmäßig erzeugt csc eine Konsolen- oder Windowsanwendung (je nach Quellcode). Wenn Sie eine Klassenbibliothek oder ein Modul erzeugen wollen, müssen Sie die target-Option mit dem entsprechenden Argument angeben. Diese Option muss vor dem Dateinamen angegeben werden. Eine Klassenbibliothek erzeugen Sie z. B. so: csc /target:library Quellcodedatei.cs.
/reference: *Dateiliste* oder /r:*Dateiliste*	Wenn Sie im Quellcode Assemblierungen (Klassenbibliotheken, Steuerelementbibliotheken etc.) verwenden, die nicht zum Standard gehören (also nicht automatisch referenziert werden) und die nicht im Ordner der erzeugten Anwendung gespeichert sind, müssen Sie diese über die reference-Option einbinden.
/lib:*Dateiliste*	Mit dieser Option können Sie zusätzliche Verzeichnisse angeben, in denen der Kommandozeilencompiler nach Assemblierungen sucht, auf die Sie verwiesen haben.

Tabelle 2.2: Die Optionen des Befehlszeilen-Compilers csc.exe

SharpDevelop

SharpDevelop ist ein recht guter Open Source-Editor für C# und VB.NET. Er ist zwar lange nicht so mächtig wie Visual Studio, bietet aber einige hilfreiche Features für das Schreiben des Quellcodes. Einen visuellen Designer für Formulare und andere sichtbare Objekte lässt SharpDevelop aber vermissen. Sie finden diesen recht intuitiv bedienbaren Editor bei *www.icsharpcode.net/opensource/sd/*. Dort können Sie auch den C#-Quellcode (!) von SharpDevelop downloaden. Die Installation des Editors ist (.NET-mäßig) einfach: Kopieren

Sie die Dateien einfach in einen beliebigen Ordner. Rufen Sie danach die Datei *SharpDevelop.exe* auf, um den Editor zu starten. Den Rest müssen Sie dann selbst herausfinden. Eine Beschreibung von SharpDevelop passt nicht in den Rahmen dieses Buchs.

2.3 Die Grundlage eines C#-Programms

Ein C#-Programm besteht mindestens aus einer Klasse, die in einer Textdatei mit der Endung *cs* gespeichert ist. Je nach Art der Anwendung unterscheidet sich der Aufbau dieser Datei.

Um hier und in den ersten Kapiteln nicht zu viel C#-Wissen vorauszusetzen, beschreibe ich zunächst lediglich, wie Sie einfache Konsolenanwendungen erzeugen. Damit Sie aber auch schon in den ersten Kapiteln mit Windowsanwendungen arbeiten können (wenn Sie wollen), zeige ich ab Seite 47, wie Sie mit Visual Studio arbeiten und wie Sie dort – ohne Grundlagenwissen – auch Windowsanwendungen erzeugen. Wenn Sie mehr über Windowsanwendungen erfahren wollen, lesen Sie Kapitel 7. Dort wird aber natürlich schon viel Grundlagenwissen vorausgesetzt.

Konsolenanwendungen

Eine Konsolenanwendung ist die einfachste Anwendung in C#. Eine solche Anwendung besteht mindestens aus einer Datei, in der eine Klasse deklariert ist. Anwendungen benötigen immer einen Einsprungpunkt (der Punkt der Anwendung, mit der diese startet). Solch einen Einsprungpunkt können Sie definieren, indem Sie einer Klasse eine Methode `Main` hinzufügen, die einen festgelegten Aufbau besitzen muss:

```
using System;

namespace Nitty_Gritty.Samples
{
   class ConsoleDemo
   {
      static void Main(string[] args)
```

```
        {
            String name;
            Console.Write("Ihr Name:");
            name = Console.ReadLine();
            Console.WriteLine("Hello World");
        }
    }
}
```

Die erste Anweisung (`using System;`) bindet den Inhalt des `System`-Namensraums so ein, dass bei der Verwendung der darin enthaltenen Typen und Namensräume `System` nicht immer wieder mit angegeben werden muss.

Die `namespace`-Anweisung erzeugt einen neuen Namensraum. Alle in den geschweiften Klammern enthaltenen Typen gehören diesem Namensraum an. Diese Anweisung ist zwar nicht unbedingt notwendig. Das Namensraum-Konzept ist aber sehr sinnvoll, weswegen Sie Ihre eigenen Anwendungen auch in Namensräumen organisieren sollten. Das Beispiel erzeugt einen übergeordneten Namensraum »Nitty_Gritty« und einen untergeordneten Namensraum »Samples«.

Die `class`-Anweisung erzeugt eine neue Klasse. In dieser Klasse wird die `Main`-Methode deklariert. Das Schlüsselwort `static` bewirkt, dass diese Methode statisch ist, d. h., ohne Instanz der Klasse aufgerufen werden kann. Das ist notwendig, da die CLR bei der Ausführung von .NET-Konsolenanwendungen keine Instanz der Klassen der Anwendung erzeugt. Das Argument dieser Methode (*args*) ist nicht unbedingt notwendig. In diesem Argument werden Befehlszeilenargumente übergeben, die ein Benutzer beim Aufruf der Anwendung eventuell angegeben hat.

In einer Konsolenanwendung können Sie über die `Write`- und die `WriteLine`-Methode der `Console`-Klasse einen Text an der Konsole ausgeben. `Write` gibt den Text an der aktuellen Position aus, `WriteLine` erzeugt zusätzlich einen Zeilenumbruch nach dem Text. Das Beispiel gibt zuerst den Text »Ihr Name:« aus. Über die `ReadLine`-Methode können Sie Eingaben an der Konsole annehmen. `ReadLine` wartet darauf, dass der Benutzer die Return-Taste betätigt und gibt die eingegebenen Zeichen als Zeichenkette zurück. Das Beispiel speichert die Ein-

gabe des Benutzers in einer Variablen (*name*), die in der ersten Anweisung der Methode (`String name;`) deklariert wird. Schließlich wird noch der Text »Hello World« in Kombination mit dem eingegebenen Namen ausgegeben.

Wenn Sie diese Quellcodedatei nun abspeichern, können Sie mit Hilfe des Kommandozeilen-Compilers *csc.exe* daraus eine Assemblierung generieren, die Sie als Programm aufrufen können.

Abbildung 2.1 zeigt die Konsole, nachdem die Anwendung erst kompiliert und danach ausgeführt wurde.

Bild 2.1: Eine Konsolenanwendung wurde erst kompiliert und dann aufgerufen

2.4 Visual Studio

Visual Studio ist die Entwicklungsumgebung von Microsoft für C++, C# und Visual Basic.NET. Diese Entwicklungsumgebung bietet Ihnen, verglichen mit anderen Entwicklungsumgebungen, die meisten Features zur Entwicklung von Programmen.

Visual Studio-Projekte und Projektmappen

Visual Studio verwaltet alle Dateien und verschiedene Einstellungen, die zu einem C#-Projekt gehören, in einer XML-Datei mit der Endung *.csproj* (*CS*harp-*Proj*ect). Über diese Datei können Sie ein C#-Projekt öffnen. Andere Projekt-Einstellungen, die aber nicht das Projekt selbst, sondern spezifische Einstellungen für Visual Basic betreffen, werden in einer Datei mit der Endung *csproj.user verwaltet*. Diese Datei können Sie gefahrlos löschen. Visual Studio erlaubt zudem, dass Sie mehrere Projekte zu einer Projektmappe zusammenfassen. Sinn-

voll ist dies z. B., wenn Sie eine Anwendung entwickeln, die mit mehreren Klassenbibliotheken arbeitet, die Sie gleichzeitig mit der Anwendung erstellen. Eine Projektmappen-Datei besitzt die Endung *.sln* (*Solution*). Wenn Sie diese Datei unter Windows ausführen, öffnet Visual Studio alle darin enthaltenen Projekte. Auch wenn die Projektmappe nur ein einziges Projekt enthält, sollten Sie dieses immer über die Projektmappen-Datei öffnen: Wenn Sie eine Projektdatei direkt öffnen, erzeugt Visual Studio ansonsten eine neue Projektmappe.

2.4.1 Der Start von Visual Studio

Visual Studio startet mit einer Startseite, die die zuletzt geöffneten Projekte anzeigt und Ihnen die Möglichkeit gibt, neue Projekte zu erzeugen oder vorhandene zu öffnen.

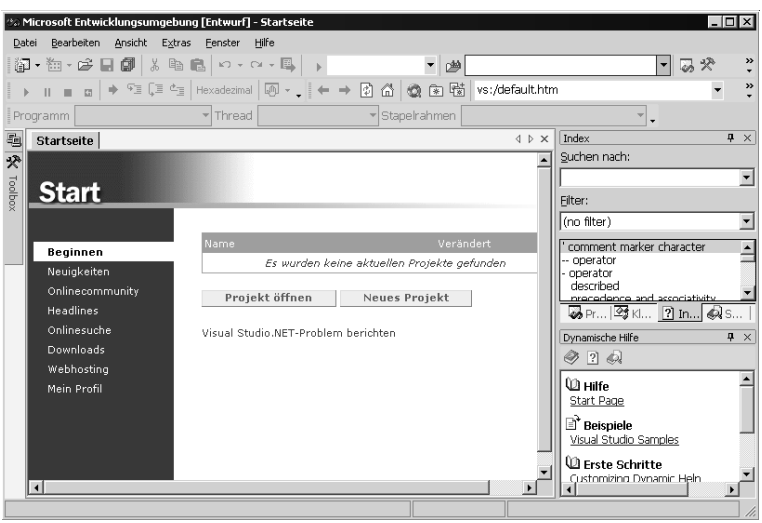

Bild 2.2: Die Startseite von Visual Studio.NET

Über den Schalter NEUES PROJEKT öffnen Sie den Dialog zur Erzeugung eines neuen Projekts. Hier können Sie für alle installierten .NET-Sprachen die verschiedenen Projekttypen (siehe Seite 41) erzeugen.

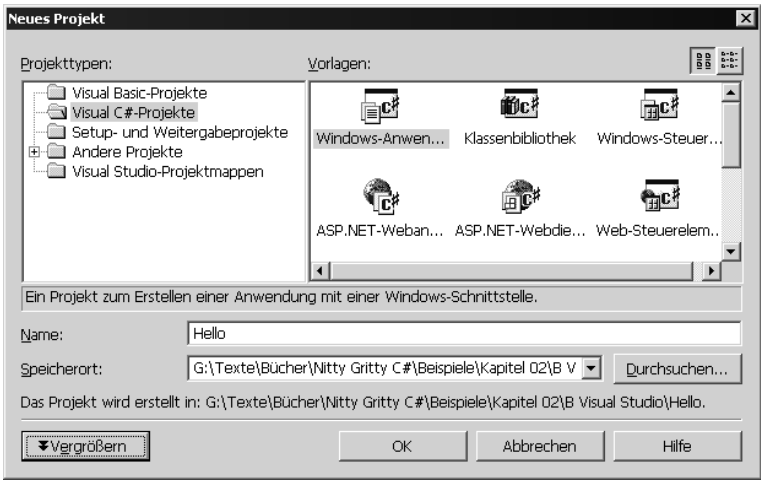

Bild 2.3: Der Visual Studio-Dialog zur Erzeugung eines neuen Projekts

Die Projekttypen, die unter dem Eintrag VISUAL C#-PROJEKTE angegeben sind, sind im Moment die für Sie interessantesten. Wählen Sie den Eintrag WINDOWS-ANWENDUNG um ein Projekt zu erzeugen, das eine Windows-Oberfläche besitzt. Bevor Sie den Dialog bestätigen, sollten Sie den Namen und den Speicherort des Projekts einstellen. Visual Studio erzeugt einen Ordner mit dem Projektnamen im Ordner, den Sie unter SPEICHERORT angegeben haben und speichert die Projektdateien in diesem Ordner. Ein nachträgliches Umbenennen der Projektdateien ist schwierig, achten Sie also auf einen korrekten Namen.

Nachdem Sie ein neues Windows-Anwendung-Projekt erzeugt haben, sieht Visual Studio etwa aus wie in Abbildung 2.4.

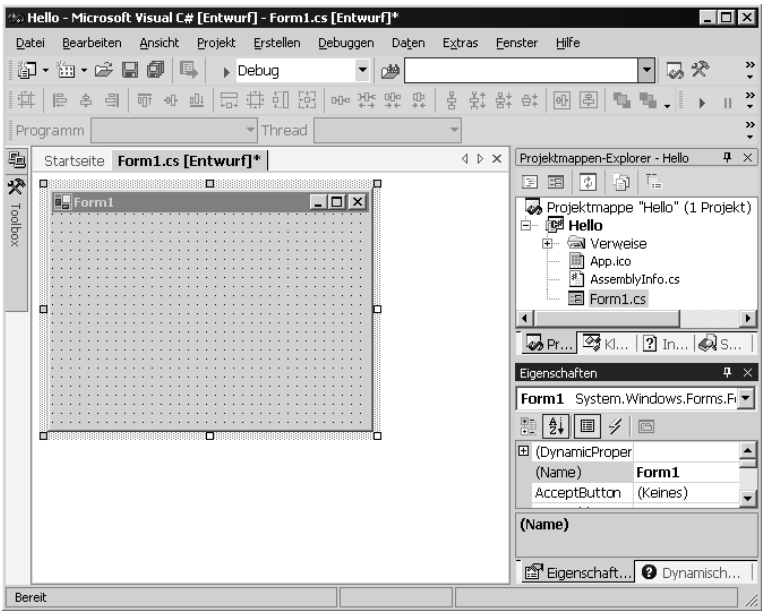

Bild 2.4: Visual Studio mit einem neuen Windowsanwendung-Projekt

2.4.2 Umgang mit den Fenstern der Entwicklungsumgebung

Die Entwicklungsumgebung besitzt viele Fenster, die teilweise aneinander andocken und teilweise registerförmig angeordnet sind. Fenster, die zur Zeit nicht sichtbar sind, können Sie über das ANSICHT- oder das DEBUGGEN-Menü sichtbar schalten. Sie können die Position der einzelnen Fenster selbst bestimmen, indem Sie diese mit der Maus an einer andere Positionen bewegen. Stellen Sie es sich aber nicht allzu einfach vor, die Fenster wieder an die Stelle zu bewegen, an der sie ursprünglich positioniert waren. Wenn Sie ein Fenster mit der Maus auf ein anderes Fenster ziehen und der Mauszeiger berührt den Symbolleistenbereich des anderen Fensters, wird das Fenster oberhalb dieses Fensters angedockt, wenn Sie es fallen lassen. Wenn Sie das Fenster etwas weiter oben fallen lassen, wird dieses in einem neuen Register unterhalb des anderen Fensters angelegt. Visual Studio zeigt dies durch eine gestrichelte Linie an, die ein Register andeutet.

*Wenn Sie die Fenster (wie ich in meinen ersten Versuchen) einmal komplett durcheinander platziert haben, gehen Sie einfach zur Startseite (*HILFE /STARTSEITE ANZEIGEN*) und wählen Sie dort unter dem Eintrag* MEIN PROFIL *einmal ein anderes und danach wieder das Visual Studio Standard-Profil aus. Dann befinden sich die Fenster wieder an der Stelle, an der sie per Voreinstellung positioniert waren.*

Die Fenster der Entwicklungsumgebung besitzen einen Pin in der Titelleiste. Wenn Sie auf diesen Pin klicken, schalten Sie die Fenster vom normalen in den versteckten Modus um. Im versteckten Modus werden die Fenster automatisch auf ein Icon verkleinert, das am Rand der Arbeitsbereich angezeigt wird. Das Toolbox-Fenster befindet sich per Voreinstellung in diesem Modus. Wenn Sie die Maus auf das Icon bewegen, wird das Fenster automatisch vergrößert. Wenn Sie einen kleinen Bildschirm besitzen, können Sie durch dieses automatische Verstecken der Fenster erreichen, dass Ihr Arbeitsbereich ausreichend groß ist. Im Menü FENSTER finden Sie einen Eintrag ALLE AUTOMATISCH AUSBLENDEN zum automatischen Verstecken aller Fenster.

Profile

Auf der Startseite von Visual Studio finden Sie unter dem Verweis MEIN PROFIL die Seite zur Einstellung eines Profils für die Entwicklungsumgebung. Hier können Sie das Tastatur-Layout, das Fenster-Layout und einen Filter für die Hilfethemen einstellen. Wenn Sie z. B. Visual Basic 6 kennen und dieses Layout bevorzugen, wählen Sie das entsprechende Tastatur- und Fenster-Layout aus. Visual Studio stellt dann sofort die Tastatur um und positioniert die Fenster der Entwicklungsumgebung entsprechend dem Profil. Ich verwende in diesem Buch nur das Visual Studio-Standard-Profil und gehe bei der Beschreibung der Menüs und Tastenkombinationen auch davon aus, dass Sie dieses Profil verwenden.

2.4.3 Die Fenster der Entwicklungsumgebung

Der Projektmappen-Explorer

Nachdem Sie ein neues Projekt erzeugt haben, sehen Sie die Dateien des Projekts im Projektmappen-Explorer (oben rechts). Dieser verwaltet die Verweise Ihres Projekts (zu externen Assemblierungen) und bildet die Dateien ab, die zum Projekt gehören. Hier erhalten Sie per Doppelklick schnellen Zugriff auf die Dateien, können diese entfernen und umbenennen. Die bei einer Windowsanwendung erzeugten Dateien erläutert die folgende Auflistung:

- *AssemblyInfo.cs*: In dieser Datei können Sie allgemeine Informationen zur später erzeugten Assemblierung für Ihr Projekt unterbringen. Dazu gehören z. B. der Titel der Assemblierung, eine Beschreibung, das Copyright und die Versionsnummer. Die hier eingegebenen Daten werden beim Kompilieren in die Metadaten der Assemblierung übernommen.
- *Form1.cs*: Diese Datei speichert ein von Visual Studio bereits in das Projekt eingefügtes Formular. Windowsanwendungen besitzen normalerweise ein Formular, mit dem die Anwendung startet. Das von Visual Studio eingefügte Formular können Sie für diesen Zweck verwenden.
- *App.ico*: Diese Datei enthält ein Icon, dass Visual Studio als Icon der Anwendung verwendet. Sie können das Icon direkt in Visual Studio bearbeiten.

Der Eintrag VERWEISE im Projektmappen-Explorer enthält die aktuell für das Projekt eingestellten Verweise.

Visual Studio hat hier bereits Verweise auf die wichtigsten Assemblierungen des .NET-Framework eingestellt. Wenn Sie andere Assemblierungen verwenden wollen, müssen Sie Verweise auf diese hinzufügen (Rechtsklick auf VERWEISE, dann VERWEIS HINZUFÜGEN). An den Stellen, bei denen ich im Buch Assemblierungen verwende auf die nicht per Voreinstellung verwiesen wird, weise ich darauf hin, dass Sie einen entsprechenden Verweis einrichten müssen.

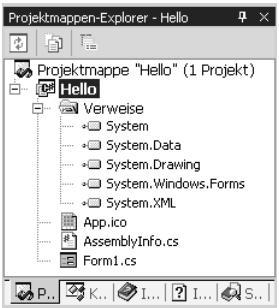

Bild 2.5: Der Projektmappen-Explorer mit aufgeklapptem Verweise-Ordner

Die Klassenansicht

Die Klassenansicht (ANSICHT / KLASSENANSICHT) stellt alle Namensräume und Klassen Ihres Projekts sehr übersichtlich dar. Wenn Sie den Eintrag einer Klasse aufklicken, sehen Sie alle Elemente dieser Klasse. Über einen Doppelklick auf einem Element gelangen Sie sehr schnell zum entsprechenden Quellcode.

Der Code-/Designer-Bereich

In der Mitte der Entwicklungsumgebung finden Sie den Code-/Designer-Bereich. Hier öffnet Visual Studio Fenster, in denen Sie den Programmcode oder die Oberfläche einer Datei bearbeiten können. Wenn Sie ein Formular oder eine andere Datei bearbeiten wollen, klicken Sie doppelt auf diese Datei im Projektmappen-Explorer. Bei Dateien, die eine Oberfläche besitzen, ziehen Sie üblicherweise Steuerelemente und Komponenten aus der Toolbox auf den Designer, bearbeiten deren Position und Größe mit der Maus und stellen deren Eigenschaften im Eigenschaften-Fenster ein. Um zu programmieren können Sie einfach auf einem Steuerelement oder auf dem Formular doppelklicken. Visual Studio öffnet dann den Code-Editor und erzeugt direkt eine neue Ereignisbehandlungsmethode für das Standardereignis. Ab Seite 61 beschreibe ich, wie Sie damit umgehen. Den Quellcode eines Formulars (oder eines beliebigen anderen Objekts mit einer Oberfläche) können Sie auch anzeigen, indem Sie das Objekt im Projekt-Explorer markieren und dann den Schalter CODE ANZEIGEN (ganz links) in der Symbolleiste des Projekt-Explorers betätigen.

Der Code-Editor

Der Code-Editor bietet eine Menge an Hilfestellungen beim Schreiben von Quellcode. Hilfreich ist – neben dem heute üblichen farblichen Hervorheben der Syntax –, dass Sie einzelne Bereiche des Quellcodes über das Plus- bzw. Minus-Symbol links auf- und zuklappen können.

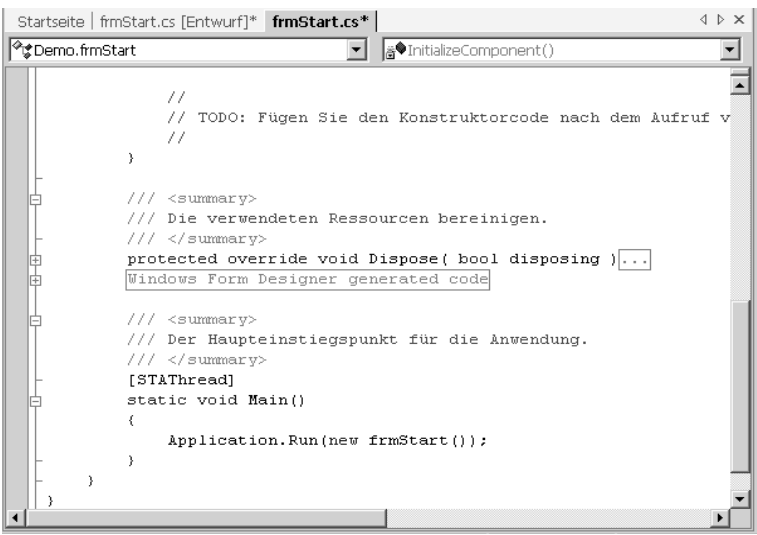

Bild 2.6: Der Code-Designer von Visual Studio mit zugeklappten Bereichen

Der Bereich WINDOWS FORM DESIGNER GENERATED CODE ist eine so genannte Region. Regionen erleichtern die Übersicht über einen Quellcode, weil Sie diese auf- und zuklappen können. Innerhalb dieser Region bringt Visual Studio alle Initialisierungen unter, die zu einem Formular gehören. Dazu gehört z. B. das Erzeugen von Steuerelementen und das Einstellen der Eigenschaften. Für den Anfang können Sie diese Region ruhig zugeklappt lassen (aber Sie halten sich ja doch nicht daran ☺). In Kapitel 7 beschreibe ich den Inhalt dieser Region.

Ein weiteres hilfreiches Feature dieses Editors ist, dass er den Quellcode automatisch einrückt. Eine Anweisung, die Sie an einer beliebi-

gen Position unterbringen, rückt der Editor meist an die korrekte Position, wenn Sie die Zeile komplettieren und mit dem Cursor verlassen. Falls das einmal nicht automatisch funktioniert, können Sie über Strg K, Strg F den aktuell markierten Text auch explizit einrücken. Wenn Sie zuvor Strg A betätigen, können Sie damit den gesamten Quellcode einer Datei korrekt einrücken.

Hilfreich ist auch, dass der Editor Syntaxfehler mit einer geschlängelten Linie unterlegt. Wenn Sie die Maus auf diese Linie bewegen, zeigt ein kleines Fenster eine Beschreibung des Fehlers an.

Der Quelltext-Editor besitzt daneben noch viele weitere hilfreiche Features. Eines davon – IntelliSense – stelle ich im nächsten Abschnitt vor. Die anderen Features finden Sie im Menü BEARBEITEN.

IntelliSense

Der Quelltext-Editor zeigt per Voreinstellung immer eine automatische Hilfe beim Schreiben von Quellcode an, die als IntelliSense bezeichnet wird. Wenn Sie z. B. einen Namensraum- oder Klassennamen gefolgt von einem Punkt schreiben, zeigt der Editor in einer Liste alle darin enthaltenen Elemente an. Wenn Sie nach dem Punkt weiterschreiben, markiert IntelliSense den passenden Eintrag in der Liste.

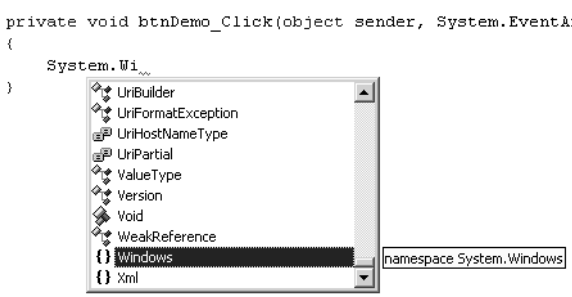

Bild 2.7: IntelliSense zeigt die Elemente des System-Namensraums an

Wenn Sie dieses Element übernehmen wollen, müssen Sie den Namen nicht zu Ende Schreiben. Schreiben Sie einfach den Text weiter, der nach dem Namen folgt (also z. B. ein Punkt oder eine Gleichheitszeichen oder eine Klammer). IntelliSense übernimmt den kompletten Text des markierten Eintrags automatisch.

Sie können dieses Feature auch nutzen, ohne den Punkt zu schreiben. Beginnen Sie einfach das Schreiben eines Bezeichners und betätigen Sie Strg Leertaste. IntelliSense zeigt dann alle im aktuellen Kontext verfügbaren Bezeichner an und positioniert die Liste auf den ersten zur Eingabe passenden Eintrag.

Ein anderes IntelliSense-Feature ist die Anzeige der Syntax einer Methode. Wenn Sie den Namen einer Methode schreiben, gefolgt von einer Klammer, zeigt IntelliSense die verschiedenen Möglichkeiten an, diese Methode aufzurufen.

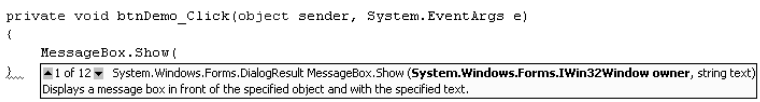

Bild 2.8: IntelliSense zeigt die Syntax einer Methode an

Mit den Cursortasten können Sie die verschiedenen Möglichkeiten durchgehen. Die `Show`-Methode der `MessageBox`-Klasse besitzt z. B. (in der mir vorliegenden Version des .NET-Framework) zwölf verschiedene Aufrufvarianten. Die `Show`-Methode der `MessageBox`-Klasse können Sie z. B. mit einer Zeichenkette im ersten Argument aufrufen. Probieren Sie das einmal selbst aus. IntelliSense müsste dann eigentlich nur noch die Varianten 7 bis 12 anzeigen, die alle eine Zeichenkette im ersten Argument erwarten.

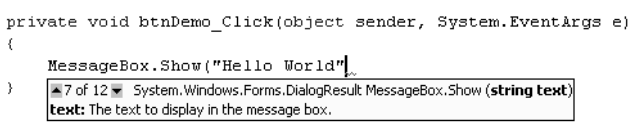

Bild 2.9: Die Version 7 der Show-Methode erwartet eine Zeichenkette im ersten Argument.

Wenn die Syntaxhilfe einmal nicht angezeigt wird, können Sie diese mit ⇧ Strg Leertaste anzeigen lassen.

Die Toolbox

Ganz links finden Sie die Toolbox. Wenn Sie die Maus auf diese Box bewegen, klappt das Toolbox-Fenster auf.

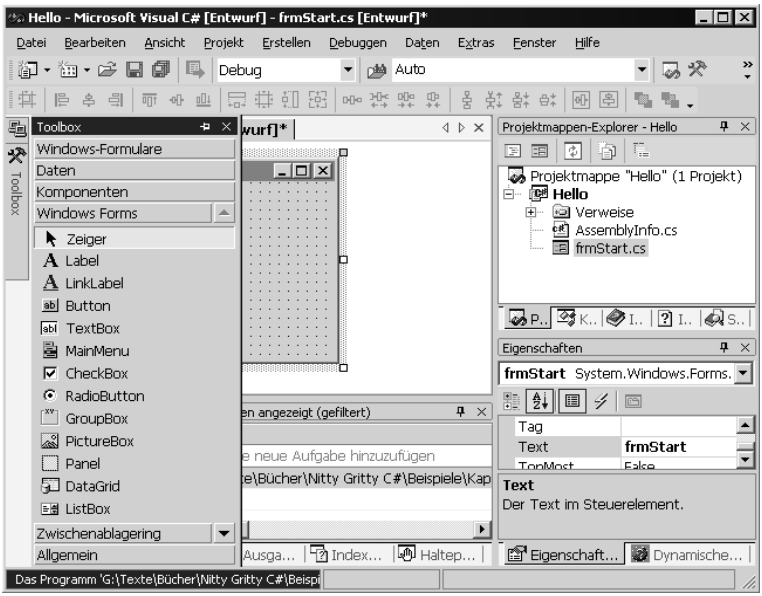

Bild 2.10: Visual Studio mit aufgeklappter Toolbox

Hier finden Sie eine große Anzahl Steuerelemente und andere Komponenten. Die Register WINDOWS-FORMS UND WINDOWS-FORMULARE enthalten Steuerelemente, die Sie auf Windows-Formularen verwenden können. Das Register KOMPONENTEN enthält spezielle Komponenten, die Sie ebenfalls auf Windows-Formularen verwenden können, die aber in der Laufzeit der Anwendung keine eigene Oberfläche besitzen. Im Register DATEN finden Sie spezielle Komponenten, die beim Datenzugriff verwendet werden (Kapitel 8). Das Register ZWISCHENABLAGERING enthält die letzten Daten, die Sie in die Zwischenablage kopiert haben. Hier können Sie auf Daten zurückgreifen, die Sie zuvor einmal in die Zwischenablage kopiert hatten.

Je nach Projektart sind einige Register der Toolbox nicht verfügbar, dafür werden andere hinzugefügt. In einem Web-Projekt ersetzt Visual Studio z. B. das Register WINDOWS-FORMULARE durch das Register WEBFORMULARE und fügt das Register HTML ein (das HTML-Steuerelemente enthält).

Wenn Sie ein Steuerelement oder eine Komponente verwenden wollen, ziehen Sie diese einfach mit der Maus auf ein Formular. Dort können Sie das Steuerelement dann mit der Maus positionieren, in der Größe verändern und dessen Eigenschaften einstellen.

Das Eigenschaftenfenster

Im Eigenschaftenfenster (unter dem Projektmappen-Explorer) stellen Sie die Eigenschaften des gerade aktiven Objekts ein. Dazu aktivieren Sie ein Objekt idealerweise mit der Maus. Wollen Sie die Eigenschaften des Formulars einstellen, klicken Sie auf einen freien Bereich des Formulars. Alternativ können Sie ein Steuerelement auch in der Objektliste des Eigenschaften-Fensters suchen und dort aktivieren. Die einzelnen Steuerelemente und das Formular besitzen eine große Anzahl an Eigenschaften. Ich beschreibe die wichtigsten in Kapitel 7. Für den Moment reicht es aus, dass Sie wissen, dass die Eigenschaft `Name` bei allen Steuerelementen und Formularen den Namen des Objekts verwaltet und die Eigenschaft `Text` die Beschriftung bzw. die eingegebenen Daten (falls es sich um ein Steuerelement handelt, das eine Beschriftung besitzt bzw. Eingaben erlaubt). Setzen Sie z. B. die Eigenschaft `Text` des Formulars auf »Hello World«, um Ihr erstes C#-Programm zu vervollständigen.

Bild 2.11: Einstellung der Eigenschaft Text eines Formulars im Eigenschaften-Fenster von Visual Studio

Testen können Sie Ihr Programm, indem Sie F5 betätigen. Viel wird allerdings noch nicht passieren. Ab Seite 61 beschreibe ich, wie Sie eine einfache Berechnung eines Bruttobetrags aus einem Nettobetrag und einem Steuerwert entwickeln.

Die dynamische Hilfe

Das Fenster für die dynamische Hilfe befindet sich ebenfalls unter dem Projektmappen-Explorer. Sie müssen normalerweise erst das Register DYNAMISCHE HILFE anklicken, um dieses Fenster anzuzeigen. Die dynamische Hilfe ist sehr hilfreich. Setzen Sie den Eingabecursor im Quellcode in ein Schlüsselwort und schon zeigt die dynamische Hilfe (meist) Verweise zu passenden Hilfethemen an. Probieren Sie es einfach aus.

Die Aufgabenliste

Die Aufgabenliste (ANSICHT / ANDERE FENSTER / AUFGABENLISTE) ist ein hervorragendes Werkzeug zur Erinnerung an Dinge, die noch zu erledigen sind. In dieser Liste können Sie eigene Aufgaben hinzufügen, indem Sie auf den Eintrag KLICKEN SIE HIER UM EINE NEUE AUFGABE HINZUZUFÜGEN klicken. Aber die Aufgabenliste kann noch mehr. So zeigt diese Liste (in der entsprechenden Ansicht) alle aktuellen Syntaxfehler des Quellcodes automatisch an. Klicken Sie mit der rechten Maustaste auf die Liste und wählen Sie im Kontextmenü unter dem Eintrag AUFGABENLISTE ANZEIGEN einen Bereich von Aufgaben, den die Liste anzeigen soll. Um die Syntaxfehler anzuzeigen wählen Sie den Eintrag BUILDFEHLER oder ALLES. Erzeugen Sie aber einmal zuvor die Assemblierung mit Strg ⇧ B, damit der Compiler alle Fehler erkennt.

Wenn Sie auf einem Eintrag doppelklicken, markiert der Code-Editor die entsprechende Stelle im Quellcode.

Ein anderes sehr hilfreiches Feature ist, dass die Aufgabenliste automatisch alle Kommentare im Quellcode, die mit »TODO«, »HACK« oder »UNDONE« beginnen, als Aufgabe anzeigt. Ein Doppelklick auf einer solchen Aufgabe markiert den entsprechenden Kommentar im Quellcode. Beachten Sie, dass das Aufgaben-Token (»TODO« etc.) im Kommentar groß geschrieben werden muss und dass Sie die Liste in der Ansicht KOMMENTARE oder ALLES anzeigen.

Die zu verwendenden Aufgaben-Token können Sie selbst verwalten. Öffnen Sie dazu die Optionen der Aufgabenliste unter EXTRAS / OPTIONEN / UMGEBUNG. In der Tokenliste können Sie die Aufgabentoken bearbeiten und neue hinzufügen. Interessant ist, dass Sie hier auch die Priorität der Aufgaben einstellen können.

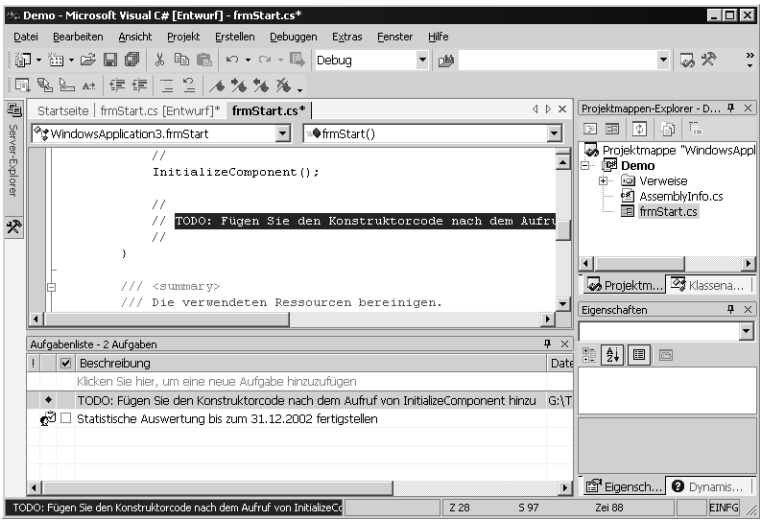

Bild 2.12: Die Aufgabenliste von Visual Studio mit einer Kommentar- und einer einfachen Aufgabe

Der Objektbrowser

Der Objektbrowser hilft dabei, die in einem Namensraum bzw. in einer Assemblierung enthaltenen Klassen und deren Elemente zu erforschen. Öffnen Sie den Objekt-Browser über das Menü ANSICHT / ANDERE FENSTER / OBJEKTBROWSER.

Der Objektbrowser zeigt den Inhalt aller Assemblierungen an, die im aktuellen Projekt referenziert werden. Beachten Sie, dass Sie viele Elemente im Eintrag *mscorlib* (das ist die Kern-Bibliothek von .NET) finden. Mit dem Objektbrowser können Sie also den Inhalt der Assemblierungen des .NET-Framework, aber auch den Inhalt externer Assemblierungen erforschen. Sehr schön ist, dass meist im unteren Bereich mehr oder weniger kurze Erläuterung des gerade aktiven Elements angezeigt wird.

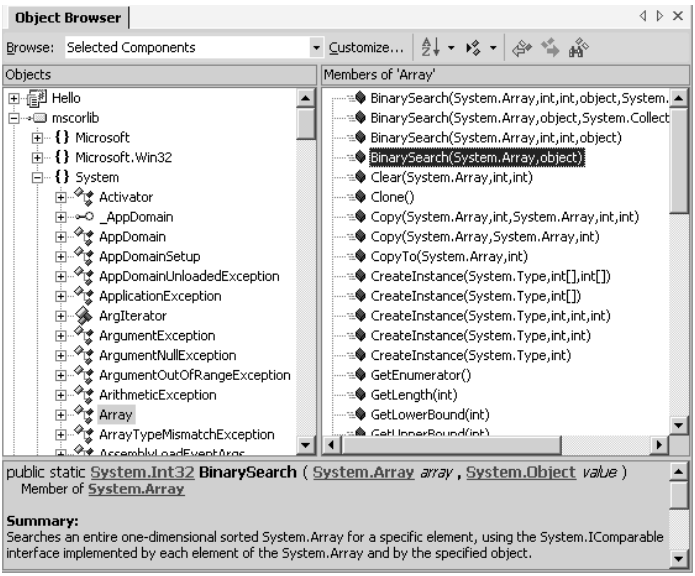

Bild 2.13: Der Objektbrowser zeigt die Deklaration der BinarySearch-Methode der Array-Klasse im System-Namensraum an

2.4.4 Ein Beispiel: Eine Netto-Brutto-Berechnung

Um die weitere Arbeit mit Visual Studio zu erläutern, verwende ich ein einfaches Programm, das aus einem gegebenen Netto- und einem Steuerwert einen Bruttowert berechnet und ausgibt. Erzeugen Sie in Visual Studio ein neues Projekt, das Sie vielleicht einfach »Bruttoberechnung« nennen.

Umbenennen des Startformulars

Vor dem Programmieren sollten Sie als Erstes das Startformular umbenennen, weil der Name »Form1« nicht besonders aussagekräftig ist. Nennen Sie dazu zunächst die Datei um. Setzen Sie den Eingabecursor auf den Namen des Formulars im Projekt-Explorer und betätigen Sie F2. Nennen Sie die Datei z. B. *StartForm.cs*. Danach müssen Sie noch den Namen des Formulars selbst umbenennen (Formulare sind Objekte und diese besitzen immer einen Namen). Klicken Sie dazu auf das Formular und suchen Sie im Eigenschaften-Fenster (un-

ter dem Projekt-Explorer) die Eigenschaft `Name`. Stellen Sie hier z. B. den Namen *StartForm* ein.

Gestalten der Oberfläche

Die Oberfläche eines Programms zur Berechnung eines Bruttowerts gestalten Sie, indem Sie verschiedene Label (für die Beschriftungen), Textboxen (für die Eingaben und die Ausgabe) und Buttons (für das Rechnen und das Beenden) auf dem Startformular platzieren. Ziehen Sie die benötigten Steuerelemente dazu zunächst auf das Formular, richten Sie diese aus und stellen Sie die Eigenschaft `Text` entsprechend Abbildung 2.14 ein.

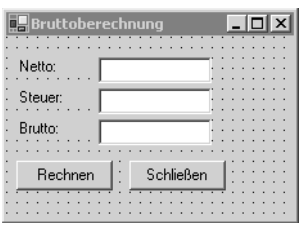

Bild 2.14: Ein Formular für eine Bruttoberechnung

Stellen Sie nun den Namen der Steuerelemente ein. Die Label bezeichne ich mit einem »lbl« als Präfix, die Textboxen erhalten den Präfix »txt«, die Buttons den Präfix »btn«. Das Label, das den Text »Netto« ausgibt, heißt z. B. »lblNetto«. Anhand der Präfixe erkenne ich später im Quellcode, worum es sich bei diesem Bezeichner handelt.

Es ist extrem wichtig, dass Sie alle Steuerelemente und das Formular benennen, bevor Sie beginnen zu programmieren. Eine spätere Änderung des Namens führt in größeren Projekten zu viel Arbeit, weil der verwendete Name auch im Quellcode angepasst werden muss. Außerdem wird der Quellcode beim nachträglichen Umbenennen inkonsistent. Der Name von über Visual Studio erzeugten Ereignisbehandlungsmethoden enthält nämlich automatisch den Namen des Objekts, für das das Ereignis gilt. Wenn Sie eine Ereignisbehandlungsmethode erzeugen, bevor Sie ein Steuerelement bzw. das Formular umbenennen, besitzt diese einen unzusammenhängenden Namen.

2.4.5 Auswerten von Ereignissen

Visual Studio macht die Auswertung von Ereignissen sehr einfach. Sie können einfach auf einem Steuerelement oder dem Formular doppelklicken, um eine Methode für das Standardereignis des Objekts zu erzeugen. Klicken Sie beispielsweise doppelt auf den Button RECHNEN, um eine Ereignisbehandlungsmethode für diesen Button zu erzeugen. Alternativ oder für andere Ereignisse können Sie das Objekt auch aus der Ereignisliste des Eigenschaften-Fensters auswählen und gegebenenfalls automatisch erzeugen. Aktivieren Sie dazu das Objekt, dessen Ereignisse Sie auswerten wollen und klicken Sie dann auf das Symbol Ereignisse im Eigenschaftenfenster. Hier können Sie den einzelnen Ereignissen eines Objekts bereits vorhandene Ereignisbehandlungsmethoden zuweisen oder einfach über einen Doppelklick auf dem Ereignisnamen neue Ereignisbehandlungsmethoden erzeugen.

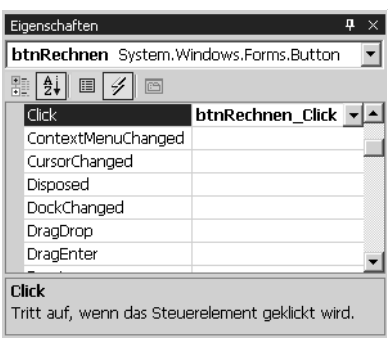

Bild 2.15: Das Eigenschaftenfenster von Visual Studio mit den Ereignissen eines Buttons

Die Ereignisbehandlungsmethode für das `Click`-Ereignis eines Buttons sieht per Voreinstellung so aus:

```
private void btnRechnen_Click(object sender,
   System.EventArgs e)
{

}
```

Die Grundlagen von Ereignisbehandlungsmethoden werden in Kapitel 7 behandelt. Für den Anfang (und meist auch für die Praxis) reicht es aus, dass Sie wissen, wie Sie mit Visual Studio eine Ereignisbehandlungsmethode erzeugen.

Die Berechnung

Für eine Bruttoberechnung benötigen Sie Zugriff auf die Eingaben. Diesen erhalten Sie über die Eigenschaft `Text` der Textboxen. Um mit den in Textform vorliegenden Eingaben rechnen zu können, müssen Sie diese zunächst in nummerische Daten konvertieren. C# ist eine typsichere Sprache und lässt mathematische Berechnungen mit Zeichenketten nicht zu. Zur Konvertierung können Sie eine der Methoden der `Convert`-Klasse verwenden. Die `ToDouble`-Methode konvertiert z. B. in einen `double`-Wert. Ich verwende zur besseren Übersicht Variablen, denen ich die konvertierten Eingaben zuweise:

```
private void btnRechnen_Click(object sender,
   System.EventArgs e)
{
   /* Deklaration von Variablen für die Berechnung */
   double netto = 0, steuer = 0, brutto = 0;
   /* Konvertieren der Eingaben in Double-Werte */
   netto = Convert.ToDouble(txtNetto.Text);
   steuer = Convert.ToDouble(txtSteuer.Text);
```

Die Berechnung erfolgt dann mit Hilfe der Variablen:

```
   /* Rechnen */
   brutto = netto * ((1 + steuer / 100));
```

Bei der Ausgabe müssen Sie wieder konvertieren, dieses Mal in einen String, da die Eigenschaft `Text` diesen Datentyp besitzt. Zu dieser Konvertierung können Sie einfach die `ToString`-Methode verwenden, die alle Datentypen besitzen:

```
   /* Ausgabe des Ergebnisses als Zeichenkette */
   txtBrutto.Text = brutto.ToString();
}
```

Prinzipiell ist die Berechnung damit fertig. Ihr Programm funktioniert aber nur dann korrekt, wenn der Anwender nummerische Daten ein-

gibt. Probieren Sie einmal aus, was passiert, wenn Sie nicht nummerische Daten eingeben und dann versuchen zu rechnen: Der Fehler wird im Debugger angezeigt. Das Debuggen behandle ich in Kapitel 6.

Für ein korrekt funktionierendes Programm müssen Sie vor der Berechnung zunächst überprüfen, ob die Eingaben vorhanden und nummerisch sind. Diese Überprüfung ist eine wichtige Grundlage der Windowsprogrammierung: Im Prinzip müssen Sie immer erst überprüfen, ob Eingaben den Anforderungen entsprechen, bevor Sie mit diesen Eingaben arbeiten. C# enthält leider keine spezifische Methode zur Überprüfung auf nummerische Eingaben. Sie können dazu aber reguläre Ausdrücke verwenden, wie ich es in Kapitel 5 beschreibe. Alternativ können Sie aber auch einfach die Ausnahme auswerten, die erzeugt wird, wenn Sie versuchen, nicht nummerische Daten in nummerische Daten zu konvertieren. Die Behandlung von Ausnahmen beschreibe ich erst in Kapitel 6. Ich zeige hier nur kurz, wie Sie diese in das Programm integrieren:

```
/* Deklaration von Variablen für die Berechnung */
double netto = 0, steuer = 0, brutto = 0;
try
{
   /* Konvertieren der Eingaben in Double-Werte */
   netto = Convert.ToDouble(txtNetto.Text);
   steuer = Convert.ToDouble(txtSteuer.Text);
   /* Rechnen */
   brutto = netto * ((1 + steuer / 100));
}
catch (FormatException)
{
   MessageBox.Show("Mindestens eine Ihrer Eingaben " +
      "ist nicht nummerisch.");
}

/* Ausgabe des Ergebnisses als Zeichenkette */
txtBrutto.Text = brutto.ToString();
```

Der `try`-Block enthält die Anweisungen, die überwacht werden sollen. Der `catch`-Block fängt Ausnahmen der Klasse `FormatException` ab, die u. a. dann ausgelöst werden, wenn eine Konvertierung nicht ausgeführt

werden kann. Mit Hilfe der `Show`-Methode der `MessageBox`-Klasse wird im Beispiel ein Text in einem Meldungsdialog ausgegeben.

Schließlich müssen Sie noch eine Ereignisbehandlungsmethode für den Schließen-Schalter erzeugen. Die Anweisung, die Sie in dieser Methode unterbringen müssen ist `this.Close();`. Die `Close`-Methode schließt ein Formular:

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

Fertig ist Ihre erste Anwendung.

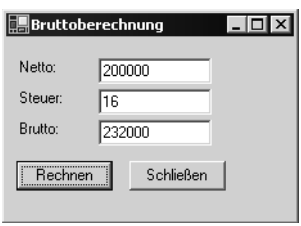

Bild 2.16: Eine Bruttoberechnung als erstes C#-Programm

Verwenden von anderen Ereignissen am Beispiel eines Längenumrechners

Das Programm zur Bruttoberechnung arbeitet mit dem `Click`-Ereignis des *Rechnen*-Buttons. Den allermeisten Programmen reicht eine Auswertung dieses Ereignisses aus. Die Steuerelemente besitzen aber sehr viele Ereignisse, die Sie für Ihre Programm auswerten können. Die wichtigsten Ereignisse werden zwar in Kapitel 7 behandelt. Ich zeigt hier aber einfach, wie Sie andere Ereignisse auswerten. Das Beispielprogramm ist ein Längenumrechner, der eine Längenangabe von Zentimeter in Inch und umgekehrt ermöglicht.

Das Programm soll immer dann, wenn der Anwender im Zentimeter-Feld die Return-Taste betätigt, den Inch-Wert berechnen und immer dann, wenn der Anwender im Inch-Feld die Return-Taste betätigt, den Zentimeter-Wert.

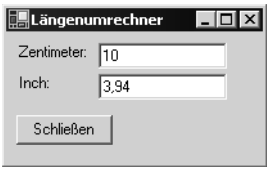

Bild 2.17: Das Formular des Längenumrechners

Erzeugen Sie in Visual Studio ein neues Projekt und nennen dies vielleicht *Längenumrechner*. Erstellen Sie das Formular. Die Textboxen habe ich *txtCentimeter* und *txtInch* genannt.

Um die Berechnung zu programmieren, verwenden Sie das `KeyPress`-Ereignis der beiden Textboxen. Suchen Sie dieses Ereignis für jede Textbox separat im Eigenschaftenfenster und klicken Sie einfach doppelt auf den Eingabebereich des Eintrags, um eine Ereignisbehandlungsmethode zu erzeugen. In dieser Methode erhalten Sie über das Argument *e* und dessen Eigenschaft `KeyChar` Informationen über die betätigte Taste. Der hier zurückgegebene Wert ist ein Zeichen, dass Sie mit '\r' (Escape-Zeichen für Return) vergleichen können. Beachten Sie die Kommentare, die einiges erläutern:

```
/* Zentimeter in Inch umrechnen */
private void txtCentimeter_KeyPress(object sender,
   System.Windows.Forms.KeyPressEventArgs e)
{
   /* Überprüfen, ob die Return-Taste betätigt wurde */
   if (e.KeyChar == '\r')
   {
      /* Variable für das Ergebnis */
      double result = 0;
      try
      {
         /* Inch-Wert berechnen */
         result = Convert.ToDouble(txtCentimeter.Text)
            / 2.54;
         /* Ergebnis formatiert ausgeben */
         txtInch.Text = result.ToString("0.00");
         /* "Trick": Default-Piepsen abschalten */
```

```
            e.Handled = true;
        }
        catch (FormatException)
        {
            /* Fehlerhafte Eingabe melden */
            txtInch.Text = "Ungültige Eingabe";
        }
    }
}

/* Inch in Zentimeter umrechnen */
private void txtInch_KeyPress(object sender, System.Windows.Forms.Key-
PressEventArgs e)
{
    /* Überprüfen, ob die Return-Taste betätigt wurde */
    if (e.KeyChar == '\r')
    {
        /* Variable für das Ergebnis */
        double result = 0;
        try
        {
            /* Zentimeter-Wert berechnen */
            result = Convert.ToDouble(txtInch.Text)
                * 2.54;
            /* Ergebnis formatiert ausgeben */
            txtCentimeter.Text = result.ToString("0.00");
            /* "Trick": Default-Piepsen abschalten */
            e.Handled = true;
        }
        catch (FormatException)
        {
            /* Fehlerhafte Eingabe melden */
            txtCentimeter.Text = "Ungültige Eingabe";
        }
    }
}
```

Das auf zwei Stellen hinter dem Komma gerundete Ergebnis habe ich übrigens mit Hilfe der `ToString`-Methode der Ergebnisvariable erreicht, indem ich dieser den Formatstring "0.00" übergeben habe. Formatierungen werden in Kapitel 5 behandelt.

2.4.6 Kompilieren und Ausführen eines Programms

Über den Befehl ERSTELLEN im gleichnamigen Menü können Sie ein Projekt kompilieren. Wenn Sie das Projekt testen wollen, können Sie dieses mit F5 oder dem Menübefehl DEBUGGEN / STARTEN erstellen und ausführen. Das Debuggen wird ausführlich in Kapitel 6 behandelt. Für den Moment reicht es aus, dass Sie eine Anwendung mit F5 starten können.

Beim Erstellen der Assemblierung verwendet Visual Studio die aktuelle Projektmappenkonfiguration, die Sie in der Projektmappenkonfigurationsliste der oberen Symbolleiste einstellen können. Per Voreinstellung ist hier die Konfiguration DEBUG eingestellt. In dieser Konfiguration werden (wiederum per Voreinstellung) Debuginformationen in die erzeugte Assemblierung generiert und der Code nicht optimiert. Eine solche Assemblierung kann (auch ohne Quellcode), mit dem Debugger bearbeitet werden, um Fehler zu finden. Die andere voreingestellte Konfiguration RELEASE kompiliert die Assemblierung ohne Debuggerinformationen (und damit ohne Ballast) und mit einer Code-Optimierung. Sie können diese Konfigurationen in den Projekteigenschaften einsehen und anpassen (wie ich es im nächsten Abschnitt beschreibe). Wenn Sie mit der Konfiguration DEBUG kompilieren, wird die Assemblierung per Voreinstellung im Unterordner *bin\debug* erzeugt. Mit der Konfiguration RELEASE erzeugt Visual Studio die Assemblierung im Ordner *bin\release*.

Tipp zu Konsolenanwendungen mit Visual Studio

Wenn Sie Konsolenanwendungen mit Visual Studio erzeugen und ausführen, sollten Sie zur Ausführung der Anwendung in Visual Studio Strg + F5 betätigen. Die Konsole wartet dann nach der Ausführung darauf, dass Sie eine Taste betätigen. Wenn Sie dagegen zum Start der Anwendung nur F5 betätigen, wird die Konsole nach der Ausführung gleich wieder geschlossen und Sie haben nur sehr wenig Zeit für die Betrachtung der Ausgaben Ihrer Anwendung.

2.4.7 Optionen der Entwicklungsumgebung und des Projekts

Eigenschaften der Entwicklungsumgebung

Die Eigenschaften der Entwicklungsumgebung erreichen Sie über das Menü EXTRAS / OPTIONEN. Wichtige Eigenschaften finden Sie im Eintrag TEXT-EDITOR. Hier können Sie u. a. die Tabulatorgröße für einzelne oder auch für alle Sprachen definieren.

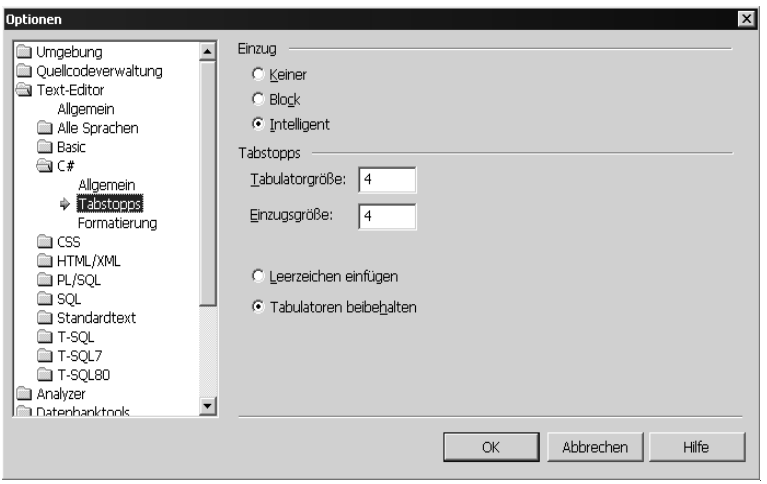

Bild 2.18: Die Optionen von Visual Studio

Interessant ist auch die Option SPEICHERORT DER VISUAL STUDIO-PROJEKTE, die Sie im Ordner UMGEBUNG unter dem Eintrag PROJEKTE UND PROJEKTMAPPEN finden. Hier können Sie den Ordner einstellen, den Visual Studio für neue Projekte als Standardordner anbietet. Eine weitere wichtige Option ist GRIDSIZE, die Sie unter WINDOWS FORMS-DESIGNER finden. In dieser Option geben Sie die Größe des Gitters an, an dem Visual Studio die Steuerelemente auf einem Formular ausrichtet. Um eine feinere Positionierung der Steuerelemente zu ermöglichen als voreingestellt, stellen Sie hier etwa den Wert 4 für beide Richtungen ein. Schalten Sie die Option SHOWGRID aus, um das Gitternetz nicht anzeigen zu lassen.

Projekteigenschaften

Jedes Projekt besitzt eigene Eigenschaften. Diese erreichen Sie, indem Sie den Befehl EIGENSCHAFTEN im PROJEKT-Menü betätigen. Dieser Befehl ist nur dann verfügbar, wenn Sie das Projekt im Projekt-Explorer aktivieren. Alternativ können Sie die Eigenschaften auch über das Kontextmenü des Projekt-Eintrags öffnen.

Sehr wichtige Einstellungen sind die, die Sie unter dem Eintrag ALLGEMEINE EIGENSCHAFTEN / ERSTELLEN finden.

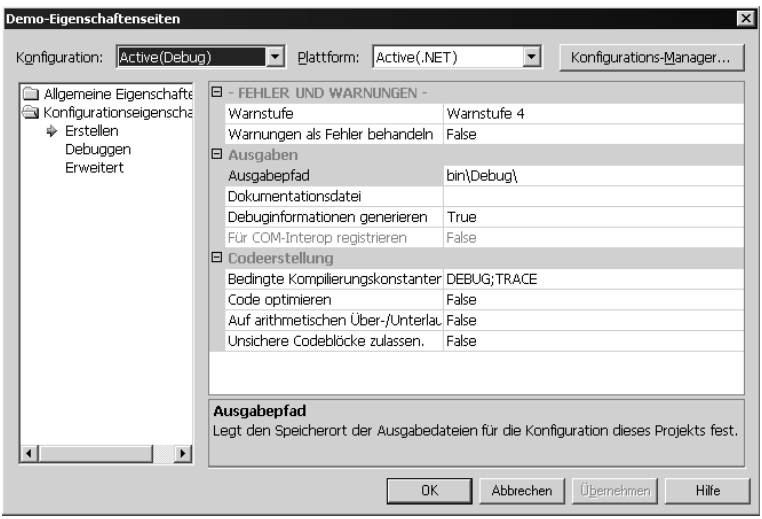

Bild 2.19: Die Projekteigenschaften eines Visual Studio-Projekts

Oben im Dialog (Abbildung 2.19) stellen Sie die Konfiguration ein, deren Einstellungen Sie ändern wollen. Per Voreinstellung stehen Ihnen die Konfigurationen *Debug* und *Release* zur Verfügung. Über den Konfigurations-Manager können Sie auch weitere Konfigurationen hinzufügen (sofern Sie diese überhaupt benötigen).

Die einzelnen Optionen werden recht gut innerhalb des Dialogs erläutert, wenn Sie diese aktivieren.

2.4.8 Weitere Features von Visual Studio

Makros und Add-Ins

Über den Menüpunkt EXTRAS / MAKROS können Sie Makros erzeugen und ausführen. Sehr interessant ist die Möglichkeit, Makros aufzuzeichnen (Strg ⇧ R). Der Makrorecorder zeichnet dann alle Schritte auf, die Sie über die Tastatur und bedingt auch über die Maus ausführen. Das aufgezeichnete Makro wird als »temporäres Makro« gespeichert, das Sie dann über den Makro-Explorer (Alt F8) umbenennen und einem Makromodul zuordnen können.

Makros sind komplette C#- oder VB.NET-Projekte, die über die Klasse `EnvDTE.DTE` auf die Entwicklungsumgebung und die darin geöffneten Projekte zugreifen. Makros können Sie also auch selbst programmieren. Die Entwicklungsumgebung kann damit sehr flexibel erweitert werden.

Add-Ins sind Makros ähnlich, nur dass diese in einer externen Assemblierung gespeichert sind und so auch an andere Entwickler verteilt werden können, ohne den Quellcode mitliefern zu müssen. Viele externe Hersteller werden Add-Ins anbieten, die Sie installieren und über den Add-In-Manager (EXTRAS / ADD-IN-MANAGER) aktivieren können. Add-Ins können Sie natürlich auch selbst entwickeln. Das soll aber nicht Thema dieses Buchs sein.

Server-Explorer

Den Server-Explorer finden Sie auf der linken Seite von Visual Studio über der Toolbox. Der Server-Explorer zeigt alle Server-Dienste an, die auf einem Server (bzw. auf dem lokalen Rechner laufen) und ermöglicht Ihnen, diese in einfacher Form zu administrieren. Sie können nicht nur den lokalen Rechner, sondern auch entfernte Server im Server-Explorer anzeigen. Interessant ist der Ordner DATENVERBINDUNGEN. Über diesen Ordner können Sie eine Verbindung zu einer Datenbank erzeugen (die aber nur für den Server-Explorer gilt). Über diese Verbindung können Sie die Datenbank dann erforschen und (je nach Datenbank) eingeschränkt auch administrieren (z. B. bei Access die Daten einer Tabelle ändern oder beim SQL-Server auch Tabellen hinzufügen und deren Struktur ändern). Mit dem Server-Explorer besitzen Sie also eine Möglichkeit, die im Rahmen einer Programment-

wicklung notwendigen Administrierungen direkt über Visual Studio ausführen zu können und notwendige Informationen ebenfalls direkt zu erhalten.

Editoren für XML, HTML, CSS, Ressourcen, Bitmaps, Cursor und Icons

Visual Studio enthält spezielle Editoren für XML-, HTML-, CSS-, Ressourcen-, Bitmap-, Cursor- und Icon-Dateien. Über den Menüpunkt PROJEKT / NEUES ELEMENT HINZUFÜGEN können Sie eine dieser Dateien dem Projekt hinzufügen. Ein Doppelklick auf der Datei im Projekt-Explorer öffnet den entsprechenden Editor.

Flexible Suche mit regulären Ausdrücken und Platzhaltern

Der Suchen- und der Ersetzen-Dialog (Strg F; Strg H) unterstützen reguläre Ausdrücke und Platzhalter (Wildcards) für das flexible Suchen und Ersetzen. Wählen Sie den entsprechenden Eintrag der Liste im Dialog. So können Sie beispielsweise alle zweistelligen Ziffern in einem Quelltext finden: »:d:d«.

Reguläre Ausdrücke werden in Kapitel 5 behandelt. Die Syntax der regulären Ausdrücke ist beim Suchen aber etwas anders als die von normalen regulären Ausdrücken. Über den HILFE-Schalter des Such- oder Ersetzen-Dialogs öffnen Sie die Hilfe für die Suche und das Ersetzen. Suchen Sie hier den Link REGULÄRE AUSDRÜCKE (bzw. REGULAR EXPRESSIONS). Über diesen Link öffnen Sie die Hilfeseite für die regulären Ausdrücke beim Suchen und Ersetzen.

Teil II

Take that!

3 Grundlagen der Sprache

Nachdem Sie in Kapitel 2 erfahren haben, wie Sie C#-Programme grundsätzlich aufbauen und kompilieren, beschreibe ich in diesem Kapitel die Grundlagen der Sprache C#. Ich zeige, wie Sie Anweisungen schreiben, mit Datentypen umgehen, mit Schleifen und Verzweigungen arbeiten und Methoden (Funktionen) schreiben. Um die Beispiele möglichst einfach zu halten, verwende ich dazu einfache Konsolenanwendungen.

Ich gehe in diesem Kapitel davon aus, dass Sie die Grundlagen der Programmierung beherrschen. Lesen Sie u. U. mein Buch »Programmieren« aus der Nitty-Gritty-Reihe, wenn Sie einzelne der hier beschriebenen Themen nicht verstehen oder die Grundlagen vertiefen wollen.

Methoden (Funktionen) werden übrigens noch nicht in diesem Kapitel, sondern erst bei der objektorientierten Programmierung im nächsten Kapitel beschrieben

3.1 Umgang mit Assemblierungen und Namensräumen

Wenn Sie die in einer Assemblierung enthaltenen Typen im Programm verwenden wollen, müssen Sie diese Assemblierung referenzieren. Die einzige Assemblierung, die nicht referenziert werden muss, weil der Compiler das automatisch macht, ist *mscorlib.dll*. Diese Assemblierung enthält die grundlegenden Typen des .NET-Frameworks in verschiedenen Namensräumen. In Visual Studio verwenden Sie zur Referenzierung anderer Assemblierungen den `Referenzen`-Eintrag im Projektmappen-Explorer, so wie ich es bereits in Kapitel 2 gezeigt habe. Wenn Sie den Kommandozeilencompiler verwenden, geben Sie die zu referenzierenden Assemblierungen im reference-Argument an.

Wenn Sie die in einem Namensraum enthaltenen Typen in Ihrem Programm verwenden wollen, können Sie diese unter Angabe des Namensraums voll qualifiziert angeben. Wenn Sie beispielsweise über die `Show`-Methode der `MessageBox`-Klasse eine Meldung ausgeben wollen, können Sie die entsprechende Anweisung so schreiben:

```
System.Windows.Forms.MessageBox.Show("Hello World");
```

Sie können den Namensraum aber auch über eine `using`-Direktive, die ganz oben im Quellcode stehen muss, quasi importieren:

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

Dann können Sie alle darin enthaltenen Typen ohne Angabe des Namensraums verwenden:

```
MessageBox.Show("Hello World");
```

Falls der Compiler dann eine Kollision mit einem Typnamen eines anderen Namensraums meldet, können Sie diesen Typ immer noch voll qualifiziert angeben.

Mit der `using`-Direktive können Sie für einen Namensraum oder eine Klasse auch noch einen Alias einrichten:

```
using MB = System.Windows.Forms.MessageBox;
```

Den Namensraum bzw. die Klasse können Sie dann über den Alias referenzieren:

```
MB.Show("Hello World");
```

Ich verwende diese Technik nicht, weil es dann schwer zu erkennen ist, welche Klasse bzw. welcher Namensraum tatsächlich verwendet wird.

Der Windows Class Viewer

Der Windows Class Viewer ist ein Tool, über das Sie die Namensräume des .NET-Framework nach Klassen mit einem bestimmten Namen durchsuchen können. Wenn Sie einmal nicht wissen, in welchem Namensraum eine Klasse verwaltet wird, suchen Sie einfach danach. Der Class Viewer zeigt nicht nur den Namensraum, sondern gleich noch die komplette Deklaration der Klasse an. Sie finden dieses Tool unter dem Namen *WinCv.exe* im Ordner *\Programme\Microsoft Visual Studio .NET\FrameworkSDK\Bin*.

3.2 Anweisungen, Ausdrücke und Operatoren

3.2.1 Anweisungen

Elementare Anweisungen

Die Mutter aller Programme, die elementare Anweisung, entspricht syntaktisch der Anweisung von C++ oder Java. Eine C#-Anweisung wird immer mit einem Semikolon abgeschlossen:

```
Console.WriteLine("Hello World");
```

Daraus folgt, dass Sie Anweisungen einfach in die nächste Zeile umbrechen können:

```
Console.WriteLine(
   "Hello World");
```

Das Umbrechen in einer Zeichenkette ist allerdings nicht ganz so einfach. Die folgende Anweisung:

```
Console.WriteLine("Hello
   World");
```

ergibt direkt mehrere Syntaxfehler. U. a. meldet der Compiler (bzw. Visual Studio) den Fehler, dass ein »(« erwartet wird.

Zeichenketten müssen prinzipiell immer in der aktuellen Zeile abgeschlossen und in einer neuen Zeile wieder begonnen werden (außer, Sie verwenden so genannte wortwörtliche Zeichenketten, wie ich es auf Seite 110 beschreibe). Wollen Sie eine Zeichenkette umbrechen (was in der Praxis sehr häufig vorkommt), schließen Sie diese ab und verbinden sie mit der Zeichenkette in der nächsten Zeile über ein +. Die folgende Anweisung ist korrekt:

```
Console.WriteLine("Hello " +
   "World");
```

Elementare Anweisungen sind, wie Sie ja wahrscheinlich bereits wissen, entweder Zuweisungen von arithmetischen oder logischen Ausdrücken an Variablen:

```
i = 1 + 1;
```

oder Aufrufe von Funktionen/Methoden:

```
Console.WriteLine("Hello World");
```

Arithmetische und logische Ausdrücke arbeiten mit den ab Seite 87 beschriebenen Operatoren. Das Beispiel verwendet die Operatoren = (Zuweisung) und + (Addition). Den Aufruf von Methoden beschreibe ich ab Seite 85.

Daneben existieren noch die Struktur-Anweisungen (Schleifen und Verzweigungen), die ich ab Seite 143 beschreibe.

Bei allen Anweisungen müssen Sie beachten, dass C# Groß- und Kleinschreibung unterscheidet. Jedes Element einer Anweisung muss genau so geschrieben werden, wie es ursprünglich deklariert wurde. Sie können z. B. nicht

```
console.writeline("Hello World");
```

schreiben, da die `Console`-Klasse und deren `WriteLine`-Methode eben genau so deklariert wurde, wie ich diese hier beschreibe.

Anweisungsblöcke

Oft müssen mehrere Anweisungen blockweise zusammengefasst werden. Eine Funktion enthält z. B. ein Block von Anweisungen:

```
private void SayHello()
{
   string name;
   Console.Write("Ihr Name: ");
   name = Console.ReadLine();
   Console.WriteLine("Hallo " + name + ", wie gehts?");
}
```


Blöcke werden in C#, wie in C++ und in Java, mit geschweiften Klammern umschlossen. Wie in anderen Sprachen auch, kann ein Block überall dort eingesetzt werden, wo eine einzelne Anweisung erwartet wird. Die `If`-Verzweigung kann z. B. mit einzelnen Anweisungen arbeiten. Die folgende Anweisung überprüft, ob heute Sonntag ist und gibt in diesem Fall einen entsprechenden Text an der Konsole aus:

```
if (System.DateTime.Now.DayOfWeek == 0)
   Console.WriteLine("Heute ist Sonntag.");
```

Wenn für einen Fall, dass heute Sonntag ist, nun aber mehrere Anweisungen ausgeführt werden sollen, müssen Sie einen Block einsetzen:

```
if (System.DateTime.Now.DayOfWeek == 0)
{
   Console.WriteLine("Heute ist Sonntag.");
   Console.WriteLine("Heute wird nicht gearbeitet.");
   Console.WriteLine("Heute gehen wir snowboarden.");

}
```

Falls Sie die letzte Anweisung nicht verstehen, weil Sie irgendwo gelesen haben, dass ich am Niederrhein wohne: Auch am Niederrhein kann man snowboarden. In Neuss. In der Allrounder-Schneesporthalle. Sogar im Sommer ☺.

Würden Sie die Blockklammern weglassen,

```
if (System.DateTime.Now.DayOfWeek == 0)
   Console.WriteLine("Heute ist Sonntag.");
   Console.WriteLine("Heute wird nicht gearbeitet.");
   Console.WriteLine("Heute gehen wir snowboarden.");
```

würde der Compiler nur die erste Anweisung der `If`-Verzweigung zuordnen und die zweite und dritte immer ausführen, auch dann, wenn kein Sonntag ist. Ich fände das zwar recht nett (jeden Tag snowboarden ...), korrekt ist die Programmierung aber nicht.

Die Blockklammern teilen dem Compiler also mit, dass die darin enthaltenen Anweisungen zusammengehören.

Sichere und unsichere Anweisungen

Normale Anweisungen sind in C# sicher. Die CLR überprüft sichere Anweisungen daraufhin, ob diese u. U. eine Operation ausführen, die Speicherbereiche beeinflussen würde, die nicht zum aktuellen Kontext gehören. Die CLR erlaubt deswegen z. B. keine Zeiger, weil diese keine Kontrolle über den Speicherbereich ermöglichen, der über den Zeiger bearbeitet wird.

Sie können jedoch auch unsichere Anweisungsblöcke verwenden, wenn Sie diese mit dem Schlüsselwort `unsafe` kennzeichnen:

```
unsafe
{
   ...
}
```

Alternativ können Sie ganze Methoden als unsicher kennzeichnen:

```
unsafe void UnsafeDemo()
{
 ...
}
```

In einem solchen Block können Sie nun alle Anweisungen unterbringen, auch solche, die die CLR als unsicher erkennen würde. Unsichere Blöcke benötigen Sie immer dann, wenn Sie direkt auf den Speicher zugreifen wollen. Normalerweise müssen Sie dies nur tun, wenn Sie extrem schnelle Programme schreiben wollen. Mit Zeigern können Sie eben direkt mit dem Speicher arbeiten. Zeiger verursachen aber auch massive Probleme, da Sie damit versehentlich Speicherbereiche überschreiben oder auslesen können, die Sie gar nicht reserviert haben. Da dieses Thema wohl eher die C++-Programmierer interessiert, die systemnahe Programme wie z. B. Treiber entwickeln, behandle ich unsichere Programmierung in diesem Buch nicht weiter.

3.2.2 Kommentare und XML-Dokumentation

C# kennt vier Arten von Kommentaren: Einfache Kommentare, mehrzeilige einfache Kommentare, Aufgabenkommentare und Dokumentationskommentare (für eine XML-Dokumentation). Unabhängig von der Art des Kommentars kompiliert der Compiler diese nicht mit in die Assemblierung.

Kommentare im Quellcode sind in den meisten Programmen der Schlüssel zum Verständnis eines Programms. Gute Programme enthalten etwa genau so viel Kommentarzeilen wie Programmzeilen. Kommentieren Sie also immer ausführlich.

Einfache Kommentare

Einfache Kommentare enthalten lediglich einen beschreibenden Text, der das Verständnis eines Programmteils erleichtert. Einfache Kommentare können Sie an das Ende einer Zeile anfügen, indem Sie diese mit zwei Schrägstrichen einleiten:

```
i = 1; // Das ist ein einfacher Kommentar
```

Mehrzeilige einfache Kommentare werden mit /* eingeleitet und mit */ beendet:

```
Console.WriteLine("Hello User"); /* Das ist ein
   mehrzeiliger Kommentar */
```

Visual Studio beginnt die zweite Zeile eines mehrzeiligen Kommentars automatisch mit einem Stern:

```
/* Das ist ein mehrzeiliger Kommentar,
 * der mit Visual Studio
 * erzeugt wurde. */
```

Die Sterne vor den einzelnen Zeilen dienen lediglich der besseren optischen Darstellung und haben keine weitere Bedeutung.

Aufgabenkommentare

Wie ich bereits in Kapitel 2 beschrieben habe, können Sie den Text eines Kommentars automatisch in die Aufgabenliste übernehmen, wenn Sie den Kommentar mit einem Aufgabentoken (per Voreinstellung: `HACK`, `TODO`, `UNDONE`, `UnresolvedMergeConflict`) beginnen:

```
static void Main(string[] args)
{
   // TODO: Den Rest programmieren :-)
}
```

Sie sollten diese speziellen Aufgabenkommentare immer dann nutzen, wenn irgendwo im Quellcode noch irgendetwas zu tun ist. Da diese Kommentare immer automatisch in der Aufgabenliste erscheinen und Sie diese Liste auch gut filtern können (etwa nur die Kommentare anzeigen lassen), haben Sie immer eine gute Übersicht über die noch zu erledigenden Aufgaben.

Dokumentationskommentare

Ein Problem normaler Kommentare ist, dass Sie daraus keine Dokumentation erzeugen können. Dieses Problem lösen Dokumentationskommentare. Aus dem Inhalt dieser Kommentare können Sie über einen speziellen Compilerschalter bzw. über eine Visual Studio- oder SharpDevelop-Option eine XML-Datei erzeugen. Diese XML-Datei können Sie dann über die Entwicklungsumgebung oder ein externes Tool in eine professionelle HTML-Dokumentation transformieren.

Ich beschreibe hier nur die Grundlagen von Dokumentationskommentaren, weil eine komplette Beschreibung den Rahmen des Buchs sprengen würde. Auf der Website finden Sie einen Artikel, der sich mit den vordefinierten XML-Elementen und der Erzeugung einer XML- und HTML-Dokumentation mit verschiedenen Tools beschäftigt.

Dokumentationskommentare werden mit drei Schrägstrichen eingeleitet. Mit diesen Kommentaren können Sie alle Typen (Klassen, Strukturen, Aufzählungen etc.) und deren Elemente (Methoden, Eigenschaften, Ereignisse) dokumentieren. Innerhalb der Kommentare können Sie einfachen Text und XML-Elemente unterbringen. Über XML-Elemente können Sie eine eigene XML-Struktur aufbauen. Sie können eigene XML-Elemente einfügen (die Sie dann aber auch selbst in der XML-Datei auswerten müssen) oder einige der vordefinierten (die von den Tools zur Erzeugung einer HTML-Dokumentation verwendet werden). Das Element *summary* wird z. B. verwendet, um einen Typen oder ein Element zusammenfassend zu beschreiben:

```
/// <summary>
/// Der Einstiegspunkt der Anwendung
/// </summary>
[STAThread]
static void Main(string[] args)
...
```

Wenn Sie das Programm mit dem Kommandozeilencompiler kompilieren, geben Sie im Argument `/doc` an, in welche Datei die XML-Dokumentation erzeugt werden soll:

```
csc /doc:Dokumentation.xml Quellcode.cs
```

In Visual Studio geben Sie die Dokumentationsdatei in den Projekteigenschaften an. Wählen Sie dazu im Kontextmenü des Projekteintrags im Projekt-Explorer den Befehl EIGENSCHAFTEN. Stellen Sie den Dateinamen unter KONFIGURATIONSEIGENSCHAFTEN / ERSTELLEN / DOKUMENTATIONSDATEI ein. Beachten Sie, dass Sie diese Einstellung für jede Konfiguration (*Debug*, *Release*) einzeln vornehmen müssen. Wenn Sie das Projekt erstellen oder mit F5 starten, wird die Dokumentationsdatei automatisch erzeugt.

Damit Sie die HTML-Dokumentation einmal ausprobieren können, erzeugen Sie eine solche ganz einfach in Visual Studio über das Menü EXTRAS / ERSTELLEN VON KOMMENTARWEBSEITEN.

3.2.3 Der Aufruf von Methoden (Funktionen)

Methoden bzw. Funktionen enthalten, wie Sie ja bereits wahrscheinlich wissen, vorgefertigte Programme, die Sie über den Namen der Methode/Funktion aufrufen können. Da Funktionen in C# nur in Klassen organisiert werden können, werden diese auch als Methoden bezeichnet. Ich verwende also den allgemeineren Begriff.

Aufruf

Methoden werden immer mit deren Namen aufgerufen, gefolgt von Klammern, in denen Sie der Methode meist Argumente übergeben, die die Ausführung steuern. Methoden sind immer in Klassen gespeichert. Die `WriteLine`-Methode gehört z. B. zu der Klasse `Console`. Deshalb müssen Sie die Klasse immer mit angeben, wenn Sie Methoden aufrufen. Eine Ausnahme ist, wenn Sie eigene Methoden aufrufen, die in der Klasse deklariert sind, in der der Aufruf erfolgt. Wie Sie ja bereits wissen, sind Klassen in Namensräumen organisiert. Wenn Sie den Namensraum nicht über die `using`-Direktive »importiert« haben, müssen Sie dessen kompletten beim Aufruf einer Methode angeben. Die `Console`-Klasse gehört z. B. zum Namensraum `System`. Den vollständigen Aufruf der `WriteLine`-Methode zeigt der folgende Quellcode:

```
System.Console.WriteLine("Hello World");
```

Binden Sie den `System`-Namensraum über `using` ein, müssen Sie nur noch die Klasse beim Aufruf angeben:

```
Console.WriteLine("Hello World");
```

Falls Sie in der OOP schon etwas bewandert sind und sich wundern, dass z. B. die `WriteLine`*-Methode der* `Console`*-Klasse so einfach aufgerufen werden kann ohne dass Sie ein Objekt dieser Klasse erzeugen: In C# ist es, wie auch in anderen Sprachen, möglich, so genannte statische Methoden (oder auch »Klassenmethoden«) in eine Klasse zu integrieren. Solche Methoden können eben ohne Instanz der Klasse aufgerufen werden. Statische Methoden beschreibe ich noch näher in Kapitel 4.*

Argumente übergeben

Die Deklaration der Methode legt fest, wie viele Argumente welchen Typs übergeben werden müssen. In Visual Studio zeigt IntelliSense beim Schreiben einer Anweisung die Deklaration(en) der Methode und damit die zu übergebenden Argumente an (Abbildung 3.1).

```
static void Main(string[] args)
{
    Console.WriteLine(
}
```

▲1 of 19 ▼ void Console.WriteLine (**string format**, params object[] arg)
format: Format string.

Bild 3.1: Die Syntaxhilfe beim Schreiben von Anweisungen mit Methoden

Viele Methoden liegen allerdings gleich in mehreren, so genannten »überladenen« Varianten vor. Das Überladen von Methoden erläutere ich noch näher für eigene Methoden ab Seite 143, fürs Erste reicht es aus, dass Sie wissen, dass eine Methode auch in mehreren Varianten vorkommen kann. Die `WriteLine`-Methode kommt z. B. (zurzeit) in 19 Varianten vor. Die einzelnen Varianten können Sie in der Syntaxhilfe über die Cursortasten oder einen Klick auf die Pfeile in der Syntaxhilfe anzeigen lassen. Die Variante 6 erwartet z. B. nur einen einfachen String (eine Zeichenkette) als Argument (Abbildung 3.2).

```
static void Main(string[] args)
{
    Console.WriteLine(
}
```

▲6 of 19 ▼ void Console.WriteLine (**string value**)
value: The value to write.

Bild 3.2: Die Variante 6 der WriteLine-Methode

Gibt die Methode einen Wert zurück, können Sie diesen Wert in Zuweisungen, in arithmetischen oder logischen Ausdrücken oder als Argument einer anderen Methode verwenden. Stellen Sie sich einfach vor, dass der Compiler Methoden immer zuerst aufruft, bevor er weitere Teile der Anweisung auswertet und das Ergebnis der Methode dann als Wert in der Anweisung weiterverarbeitet. So sind einfache Zuweisungen möglich:

```
sinY = Math.Sin(y);
```

oder Ausdrücke:

```
result = Math.Min(x1, x2) / Math.Max(y1, y2);
```

oder geschachtelte Aufrufe:

```
result = Math.Min(y1, Math.Max(y2, y3));
```

Überall da, wo ein bestimmter Datentyp erwartet wird, können Sie neben Konstanten und Variablen immer auch eine Methode einsetzen, die diesen Datentyp zurückgibt.

3.3 Datentypen

Wie Sie ja sicherlich bereits wissen, kommen Datentypen in einem Programm sehr häufig vor. Wenn Sie einen Ausdruck schreiben, verwendet dieser einige Datentypen und besitzt im Ergebnis auch einen Datentyp. Variablen besitzen einen Datentyp, einfache Wertangaben ebenfalls, genau wie Argumente und Rückgabewerte von Methoden. Der Umgang mit Datentypen ist also sehr wichtig und wird deshalb hier grundlegend beschrieben.

C# unterstützt die üblichen Datentypen wie z. B. `int` (Integer) und `double`. Alle C#-Typen sind nur Aliasnamen für Datentypen, die in der CLR definiert sind. `int` steht z. B. für `System.Int32`. Prinzipiell können Sie immer die CLR-Typen verwenden, z. B. um eine Variable zu deklarieren:

```
System.Int32 i;
```

Verwenden Sie aber lieber die C#-Typen, damit Ihr Quellcode besser lesbar wird.

3.3.1 Typsicherheit, der Datentyp object, Werte- und Referenztypen

Typsicherheit

C# ist eine typsichere Sprache. Wenn irgendwo ein bestimmter Typ erwartet wird, können Sie nur einen Typ einsetzen, der zum erwarteten passt. Die Zuweisung eines Strings an eine `int`-Variable ist z. B. nicht möglich:

```
string s = "10";
int i = s; // Fehler
```

Das Beispiel resultiert im Compilerfehler »Implizite Konvertierung des Typs 'string' zu 'int' nicht möglich«. C# sichert damit ab, dass Sie nicht versehentlich einen falschen Datentyp verwenden. Wollen Sie trotzdem einen anderen Datentyp einsetzen als der Compiler erwartet, müssen Sie diesen explizit konvertieren. Wie das geht, zeige ich ab Seite 113. Hier nur ein Beispiel:

```
int i = Convert.ToInt32(s);
```

Ist der zugewiesene Datentyp vom Typ her passend und kleiner oder gleich groß, konvertiert C# diesen meist automatisch. Die Zuweisung eines `byte`-Typs an einen `int`-Typ ist z. B. problemlos möglich, weil `byte` immer in einen `int`-Typ hineinpasst:

```
byte b = 10;
int i = b;
```

Alles ist ein Objekt

In C# sind alle Datentypen Klassen, die zumindest von der Klasse `object` abgeleitet sind. Datentypen sind also immer Objekte und besitzen zumindest die von `object` geerbten Instanzmethoden. Instanzmethoden sind Methoden, die nur über eine Instanz der Klasse aufgerufen werden können.

Methode	Beschreibung
`Equals(object` *obj*`)`	vergleicht die Daten eines Objekts mit den Daten eines anderen Objekts auf Gleichheit (d. h., dass die in den Objekten gespeicherten Werte gleich sind).

Methode	Beschreibung
`GetHashCode()`	ermittelt einen Hash-Code für das Objekt. Ein Hash-Code ist ein aus den Daten des Objekts ermittelter Code, der das Objekt in verkürzter Form identifiziert.
`GetType()`	ermittelt den Typ des Objekts. Gibt ein Objekt der Klasse `Type` zurück. Aus diesem Objekt können Sie sehr viele Informationen auslesen. Die Eigenschaft `Name` liefert z. B. den Namen des Typs, die Eigenschaft `Namespace` gibt den Bezeichner des Namensraums zurück, in dem der Typ verwaltet wird.
`ToString()`	Diese Methode gibt die Daten des Objekts als Zeichenkette zurück.

Tabelle 3.1: Die Instanzmethoden des Basistyps object

In den von `object` abgeleiteten Typen werden diese Methoden normalerweise mit einer neuen Implementierung überschrieben. Die `ToString`-Methode gibt z. B. bei einem `int`-Datentyp die gespeicherte Zahl als Zeichenkette zurück:

```
int i = 10;
Console.WriteLine(i.ToString());
```

Bei einem `double`-Typ wird die Zeichenkette unter Berücksichtigung einer Ländereinstellung zurückgegeben. `ToString` können Sie ohne Argument aufrufen, dann wird die Systemeinstellung verwendet:

```
double x = 1.234;
Console.WriteLine(x.ToString());
```

Auf deutschen Systemen kommt dabei der String »1,234« heraus. Alternativ können Sie diese Methode bei einem `double`-Wert auch mit einem Format-String aufrufen:

```
Console.WriteLine(x.ToString("0.00"));
```

Die Zahl wird nun auf zwei Stellen hinter dem Komma formatiert ausgegeben. Den Umgang mit Zeichenketten und deren Formatierung beschreibe ich in Kapitel 5.

Die Verwaltung aller Datentypen als Objekt ist sogar so konsequent, dass Sie für einfache Konstanten auch die Methoden der Klasse aufrufen können:

```
string s = 10.ToString();
```

Die einzelnen Typen besitzen meist noch zusätzliche Methoden und Eigenschaften. Der Typ `string` besitzt z. B. eine Vielzahl an Methoden zur Arbeit mit Zeichenketten, wie die `Replace`-Methode, über die Sie einen Teilstring durch einen anderen String ersetzen können.

Einige Methoden sind statisch. Diese Methoden können Sie aufrufen, ohne eine Instanz des Datentyps zu besitzen. Die `Format`-Methode der `String`-Klasse ist z. B. eine solche statische Methode:

```
Console.WriteLine(String.Format("{0:0.00}", 1.234));
```

Ich beschreibe die wichtigsten dieser Methoden in Kapitel 5.

Werte- und Referenztypen

C# unterscheidet Werte- und Referenztypen. Wertetypen speichern ihren Wert direkt im Stack, der Wert von Referenztypen wird auf dem Heap gespeichert. Ein Referenztyp referenziert seinen Wert. Zu den Wertetypen gehören alle Standard-Datentypen (außer `string`), Strukturen und Aufzählungen. Alle anderen Typen, wie »richtige« Objekte und der Typ `string`, sind Referenztypen.

Der Stack ist ein spezieller Speicherbereich, den der Compiler für jede aufgerufene Methode neu reserviert. Alle lokalen Daten einer Methode werden, sofern es sich um Wertetypen handelt, immer auf dem Stack abgelegt. Der Stack wird auch verwendet, um Argumente an eine Methode zu übergeben. Der Aufrufer legt die Argumente auf dem Stack ab, die Methode liest die Argumente aus dem Stack aus.

Der Heap ist ein anderer Speicherbereich, der allerdings global für das gesamte Programm gilt und so lange besteht, wie das Programm läuft. Auf dem Heap werden üblicherweise programmglobale Daten, aber eben auch Referenztypen abgelegt.

Wenn eine Variable einen Wertetyp besitzt, speichert diese Variable den Wert (auf dem Stack). Eine Variable, die einen Referenztyp besitzt, speichert nicht den Wert, sondern nur die Adresse des Speicherbereichs, der den Wert verwaltet. Die Adresse wird auf dem Stack verwaltet, der Wert auf dem Heap.

Weil Wertetypen direkt im Stack gespeichert sind, kann der Compiler den Wert dieser Typen sehr schnell bearbeiten. Um auf einen Referenztyp zuzugreifen, benötigt der Compiler also zwei Lesevorgänge: Einmal muss die Adresse der Daten aus dem Stack ausgelesen werden, dann kann der Compiler mit dieser Adresse die Daten selbst referenzieren. Referenztypen werden also prinzipiell etwas langsamer bearbeitet als Wertetypen. Referenztypen besitzen aber auch Vorteile.

Wenn Sie einen Wertetyp an eine Methode übergeben, erzeugt der Compiler immer eine Kopie der gespeicherten Daten. Die Übergabe von Wertetypen, die große Datenmengen speichern, kann somit recht viel Zeit in Anspruch nehmen. Übergeben Sie einen Referenztyp, wird nur die Referenz auf die gespeicherten Daten übergeben, was nur sehr wenig Zeit benötigt.

Auf Referenztypen können mehrere Referenzen zeigen. Das ergibt sich schon daraus, dass nur die Referenzen bei Zuweisungen oder bei der Übergabe eines Referenztyps an eine Methode kopiert werden. Diese Technik wird bei der objektorientierten Programmierung sehr häufig verwendet. Mit Wertetypen ist das nicht möglich, weil der Compiler immer die Werte kopiert.

Wenn Sie selbst Typen erzeugen, können Sie eigentlich nur bei den strukturierten Typen entscheiden, ob Sie einen Werte- oder einen Referenztyp verwenden: Entweder Sie erzeugen eine Struktur (Wertetyp) oder eine Klasse (Referenztyp). Wie das geht, zeige ich in Kapitel 4.

Erzeugen von Wertetypen

Wertetypen werden implizit erzeugt, wenn Sie diese deklarieren. Das folgende Beispiel deklariert eine Integer-Variable:

```
int i;
```

Der Compiler reserviert sofort einen Speicherbereich, der allerdings noch uninitialisiert ist. Bevor Sie mit dieser Variable arbeiten können, müssen Sie einen Wert zuweisen. Ansonsten erzeugt der Compiler einen Kompilierfehler wenn Sie eine solche Variable im Programm lesen wollen. Die Initialisierung können Sie direkt bei der Deklaration erledigen:

```
int i = 0;
```

Der Compiler sichert damit ab, dass Wertetypen immer einen definierten Wert speichern.

Erzeugen von Referenztypen

Wenn Sie einen Referenztypen deklarieren, verweist dieser noch nicht auf eine Instanz des Typs:

```
RefType r;
```

Genau wie bei Wertetypen können Sie uninitialisierte Referenztypen nicht verwenden, der Compiler meldet dann einen Fehler. Sie können dem Typ aber den Wert `null` zuweisen. `null` steht dafür, dass eine Referenz auf keine Instanz zeigt:

```
RefType r = null;
```

Beim Versuch, mit einem solchen Typ zu arbeiten, generiert das Programm dann allerdings eine Ausnahme. Sie können allerdings abfragen, ob ein Referenztyp auf eine Instanz zeigt, indem Sie mit `null` vergleichen:

```
RefType r = null;
if (r != null)
   r.Value = 10;
else
   Console.WriteLine("r zeigt nicht auf eine Instanz");
```

Möglicherweise werden Sie sich fragen, was das Ganze soll. In der Praxis kommt es aber sehr häufig vor, dass Referenztypen nicht auf eine Instanz zeigen. Gut, wenn Sie dann damit umgehen können.

Wenn Sie Referenztypen verwenden wollen, müssen Sie diese über das Schlüsselwort `new` erzeugen:

```
RefType r = new RefType();
```

In den Klammern können Sie Argumente übergeben, die den Typen direkt bei der Erzeugung initialisieren. Welche Werte übergeben werden können, legen die Konstruktoren dieses Typs fest. Konstruktoren werden in Kapitel 4 behandelt.

Zerstören von Typen

Irgendwann einmal muss jeder Datentyp sterben. Spätestens dann, wenn ein Programm beendet ist, werden alle Speicherbereiche freigegeben, die das Programm reserviert hatte. Typen, die Sie in Methoden verwenden, werden automatisch freigegeben, wenn die Methode beendet ist und die Variablen bzw. Referenzen ungültig werden. Wertetypen werden, da sie auf dem Stack angelegt werden, immer sofort nach der Ausführung der Methode freigegeben, da der Compiler dann den gesamten Stack freigibt.

Referenztypen, die ja auf dem Heap angelegt werden, werden vom Garbage Collector automatisch zerstört. Sie brauchen also nichts weiter zu tun, als die Referenzen auf ein Objekt freizugeben bzw. sich darauf zu verlassen, dass diese automatisch freigegeben werden, wenn die Objektvariable ihren Gültigkeitsbereich verlässt. Wenn Sie das Objekt selbst freigeben wollen setzen Sie die Objektvariable auf `null`:

```
p = null;
```

Dieses explizite Freigeben wird allerdings nur in sehr seltenen Fällen sinnvoll sein. Wenn Sie z. B. in einer aufwändigen Funktion Objekte einsetzen und nach deren Verwendung Arbeitsspeicher sparen wollten, während die Funktion noch weiter ausgeführt wird, könnten Sie auf die Idee kommen, die Objektreferenzen von Hand freizugeben. Dummerweise werden diese aber erst aus dem Arbeitsspeicher entfernt, wenn der Garbage Collector Zeit dafür hat. Und das tritt mit ziemlicher Sicherheit nicht mitten in der Ausführung einer Funktion ein.

Zuweisung von Typen

Wenn Sie einen Wertetyp einem anderen zuweisen, werden die Werte kopiert:

```
int i, j;
i = 10;
j = i;
```

Am Ende dieses Beispiels besitzt `j` den Wert 10, ist aber immer noch ein eigener Speicherbereich. Wenn Sie `j` danach verändern, wird `i` nicht davon beeinflusst:

```
j = 11; // i ist immer noch 10
```

Wenn Sie einen Referenztypen einem anderen Referenztypen zuweisen, werden üblicherweise nicht die Werte, sondern die Referenzen kopiert:

```
public class RefType
{
   public int Value;
}
public class Start
{
   [STAThread]
   static void Main(string[] args)
   {
   /* Referenztypen */
      RefType r1 = new RefType();
      RefType r2 = new RefType();
      r1.Value = 10;
      /* die Referenz von r2 wird auf die
       * Referenz von r1 gesetzt */
      r2 = r1;
      /* eine Änderung von r2 betrifft nun auch r1 */
      r2.Value = 11;
      Console.WriteLine("r1 = " + r1.Value +
         ", r2 = " + r2.Value);
   }
}
```

In der letzten Anweisung dieses Beispiels besitzt `r1` denselben Wert wie `r2`, beide Referenzen zeigen auf denselben Speicherbereich im Heap. Der Speicherbereich, auf den `r2` anfangs gezeigt hat, wurde vom Garbage Collector zwischenzeitlich entsorgt, da er nicht mehr referenziert wird.

Der Typ string als Ausnahme

`String` ist eigentlich auch ein Referenztyp. Der Compiler erlaubt für diesen Typ aber auch eine Erzeugung ohne `new`:

```
String s1 = "Das ist ein String";
```

Alternativ können Sie den String auch mit `new` erzeugen. Dem Konstruktor können Sie dazu verschiedene Argumente übergeben. U. a. können Sie den String mit einer bestimmten Anzahl von Zeichen initialisieren:

```
String s2 = new String('*', 1024);
```

Bei Zuweisungen eines Strings auf einen anderen verhält sich ein String ebenfalls etwas anders als ein normaler Referenztyp: Wenn Sie zwei Stringvariablen oder Eigenschaften einander zuweisen, kopiert der Compiler zunächst die Referenz.

```
String s1, s2;
s1 = "Zaphod";
s2 = s1;
```

Nach der Ausführung dieser Anweisungen referenziert *s2* dieselbe Zeichenkette wie *s1*. Soweit stimmen Strings noch mit normalen Referenztypen überein.

Wenn Sie einen String über die zweite Variable bearbeiten, erzeugt der Compiler einen neuen String:

```
s2 = "Ford";
```

Nun referenziert *s1* einen anderen String als *s2*. Eine Änderung des Strings in *s2* bewirkt keine Änderung des Strings in *s1*. C# sorgt damit dafür, dass Strings ähnlich wie Wertetypen behandelt werden können, obwohl es sich um Referenztypen handelt.

Wertetypen sind Objekte? Oder: wie werden Objekte gespeichert?

Vielleicht fragen Sie sich, warum Wertetypen auch Objekte sind, obwohl der Compiler diese im Stack anlegt. Wo sind dann die Methoden dieser Typen gespeichert? Die Antwort darauf gibt die Beschreibung der allgemeinen Speichertechnik für Objekte.

Wenn der Compiler Objekte im Speicher anlegt, werden die Methoden (und die statischen Eigenschaften) nicht im Speicherbereich des

Objekts gespeichert. Für die Objekte einer Klasse speichert der Compiler diese in einem separaten Speicherbereich, der nur einmal pro Klasse angelegt wird. Das unterscheidet sich nicht bei Werte- und Referenztypen. Abbildung 3.3 zeigt die Speichertechnik am Beispiel von Objekten einer imaginären Circle-Klasse zur Verwaltung von Kreisdaten:

```
public class Circle
{
   public int Radius;
   public int Color;
   public static
   public double GetSurface()
   {
      return Radius * Radius * PI;
   }
}
```

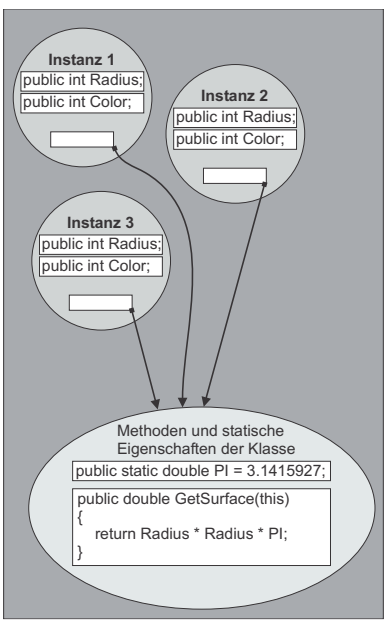

Bild 3.3: Der Compiler speichert die Methoden und die statischen Eigenschaften einer Klasse immer separat

Die einmalige Speicherung der Methoden und statischen Eigenschaften reduziert den Speicherbedarf von Objekten. Die Instanz der Klasse ist der Speicherbereich für die einzelnen Objekte, in dem der Compiler die Eigenschaften anlegt, die keine statischen Eigenschaften sind. Irgendwo verwaltet der Compiler für die einzelnen Typen eine Referenz auf den Speicherbereich der Klasse. Für Wertetypen wird der Compiler diese Referenz wahrscheinlich separat verwalten, für Referenztypen innerhalb der Instanz. Abbildung 3.3 deutet dies über die Felder an, die auf den Klassen-Speicherbereich zeigen.

Wenn Sie nun eine Methode eines Typs aufrufen, kann der Compiler diese über den Zeiger auf den Klassen-Speicherbereich lokalisieren. Um innerhalb der Methode einen Bezug zur Instanz der Klasse zu haben, übergibt der Compiler der Methode ein zusätzliches Argument mit dem Namen `this`, das auf die Instanz zeigt. Über diese Referenz kann die Methode auf die Eigenschaften des Objekts zugreifen.

3.3.2 Übersicht über die Standarddatentypen

Die vordefinierten C#-Datentypen listet Tabelle 3.2 auf. Die C#-Bezeichner dieser Datentypen sind nur Aliase für den entsprechenden CLR-Datentyp. `int` steht z. B. für den Typ `System.Int32`. Die CLR-Datentypen sind im Namensraum `System` gespeichert. Beachten Sie, dass die vorzeichenlosen Integertypen nicht CTS-kompatibel sind. Wenn Sie Komponenten entwickeln, die von anderen .NET-Programmiersprachen verwendet werden sollen, sollten Sie diese Typen nur für interne, private Zwecke verwenden.

C#-Datentyp	CLR-Datentyp	Größe	Wertebereich
`sbyte`	`SByte`	8 Bit	-128 bis 127
`byte`	`Byte`	8 Bit	0 bis 255
`char`	`Char`	16 Bit	ein beliebiges Unicode-Zeichen
`short`	`Int16`	16 Bit	-32.768 bis 32.767
`ushort`	`UInt16`	16 Bit	0 bis 65.535
`int`	`Int32`	32 Bit	-2.147.483.648 bis 2.147.483.647

C#-Datentyp	CLR-Datentyp	Größe	Wertebereich
`uint`	`UInt32`	32 Bit	0 bis 4.294.967.295
`long`	`Int64`	64 Bit	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
`ulong`	`UInt64`	64 Bit	0 bis 18.446.744.073.709.551.615
`float`	`Single`	32 Bit	$\pm 1.5 \times 10^{-45}$ bis $\pm 3.4 \times 10^{38}$ mit maximal 7 Dezimalstellen
`double`	`Double`	64 Bit	$\pm 5.0 \times 10^{-324}$ bis $\pm 1.7 \times 10^{308}$ mit maximal 15-16 Dezimalstellen
`decimal`	`Decimal`	128 Bit	1.0×10^{-28} bis 7.9×10^{28} mit maximal 28-29 Dezimalstellen
`bool`	`Boolean`	8 Bit	`true` oder `false`
`string`	`String`	variabel	beliebige Unicode-Zeichenketten

Tabelle 3.2: Die C#-Standarddatentypen

Instanzmethoden der Datentypen

Die Standardtypen besitzen wie alle Typen die Instanzmethoden, die sie von `object` geerbt haben (Seite 88). Der `ToString`-Methode können Sie bei nummerischen Typen allerdings auch einen Format-String oder einen Format-Provider übergeben, um die Zahl zu formatieren. Formatierungen werden in Kapitel 5 behandelt. Ich zeige hier nur kurz, wie Sie z. B. einen `double`-Wert auf zwei Stellen hinter dem Komma formatieren können:

```
double d = 1.2345;
Console.WriteLine(d.ToString("0.00"));
```

Neben den geerbten Methoden besitzen die Standardtypen noch die in Tabelle 3.3 dargestellten zusätzlichen Instanzmethoden.

Methode	Beschreibung
`int CompareTo(` `object value)`	vergleicht einen Typ mit einem anderen. Rückgabe: < 0: der Typ ist kleiner als der andere, 0: beide Typen sind gleich, > 0: der Typ ist größer als der andere.
`TypeCode GetTypeCode()`	ermittelt den Datentyp des Typs. Gibt einen Wert des Typs `TypeCode` zurück. `TypeCode` ist eine Aufzählung von Konstanten, die so benannt sind wie die Typen. `TypeCode.Byte` definiert z. B. einen Byte-Typ.

Tabelle 3.3: Die zusätzlichen Methoden der Standarddatentypen

Die `CompareTo`-Methode wird hauptsächlich implizit verwendet, wenn ein Typ in einer Auflistung gespeichert ist, die sortierbar ist. Explizit brauchen Sie diese Methode eigentlich nie aufrufen, da Sie dazu für Vergleiche die Vergleichsoperatoren (Seite 135) verwenden können.

Der Datentyp `String` besitzt noch eine Vielzahl weiterer Methoden. Ich beschreibe die wichtigsten davon separat in Kapitel 5.

Klassenmethoden und -eigenschaften der Datentypen

Alle Datentypen besitzen neben den Instanzmethoden auch noch Klassenmethoden und -eigenschaften, die Sie ohne Instanz der Klasse verwenden. Die `MaxValue`-Eigenschaft der nummerischen Typen gibt z. B. den größten speicherbaren Wert zurück:

```
Console.WriteLine("double kann maximal " +
   double.MaxValue + " speichern.");
```

Über die `Parse`-Methode können Sie einen String in den entsprechenden Typ umwandeln:

```
d = double.Parse("1.234");
```

Tabelle 3.4 zeigt die wichtigsten dieser Eigenschaften und Methoden.

Eigenschaft/ Methode	Beschreibung
`MinValue`	liefert den kleinsten speicherbaren Wert
`MaxValue`	liefert den größten speicherbaren Wert
Typ `Parse(` `string` *s* `[...])`	Über diese Methode können Sie einen String in den Typ umwandeln, auf dem Sie die Methode anwenden. `Parse` erlaubt bei nummerischen Typen zusätzlich die Angabe der zu verwendenden Kultur. Die Kultur wird in Kapitel 5 behandelt.

`bool TryParse(`
`string` *s*`,`
`NumberStyles`
style`,`
`IFormatProvider`
provider`,`
`out double`
result`)`
	Über diese komplexe Methode können Sie überprüfen, ob eine Umwandlung eines Strings in einen speziellen Typ möglich ist.

Tabelle 3.4: Die wichtigsten Klasseneigenschaften und -methoden der Standardtypen

Die `TryParse`-Methode will ich hier nicht näher beschreiben, weil Sie dazu wissen müssen, wie Sie mit Schnittstellen umgehen (`IFormatProvider`) und was Kultur-Informationen sind. Schnittstellen werden in Kapitel 4 behandelt, Kultur-Informationen (Globalisierung) in Kapitel 5. Das folgende Beispiel überprüft einen String darauf, ob der mit den aktuellen Ländereinstellungen in einen `double`-Wert konvertierbar ist:

```
string s = "abc";
System.Globalization.CultureInfo culture;
culture =
   System.Globalization.CultureInfo.CurrentCulture;
if (!double.TryParse(s,
   System.Globalization.NumberStyles.Float,
   culture.NumberFormat, out d))
   Console.WriteLine(s + " kann nicht in einen " +
      "double-Wert umgewandelt werden.");
```

3.3.3 Integerdatentypen

C# unterscheidet einige Datentypen für die Speicherung von Integerwerten (Werte ohne Dezimalanteil): `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` und `ulong`. Bei der Auswahl dieser Datentypen müssen Sie eigentlich nur beachten, dass der Datentyp groß genug ist für Ihre Anwendung und ob Sie ein Vorzeichen benötigen. Machen Sie sich aber nicht zu viel Gedanken: Wenn Sie für einzelne Daten einen zu großen Datentyp verwenden, macht das dem Programm nichts aus. Der Arbeitsspeicher ist heute ja wohl immer groß genug. Verwenden Sie lieber einen größeren Typ, damit Sie später bei der Speicherung von Daten keine Probleme haben. Wenn Sie z. B. für die Speicherung einer Kundennummer `ushort` verwenden, und die Nummer wird größer als 65.535, gibt es Probleme (siehe unter »Über- und Unterläufe« auf Seite 105). Achten Sie aber auf die Größe, wenn Sie sehr viele Daten speichern (z. B. in einem Array).

Die gängigen Literale für Integerwerte sind Zahlen:

```
int i = 1234;
```

Alternativ können Sie auch die hexadezimale Schreibweise verwenden. Stellen Sie dazu ox vor den Wert:

```
byte i = 0xff; // = 255
```

Wenn Sie eine Zahl ohne Dezimalstelle schreiben, erkennt der Compiler diese als `int`, `uint`, `long` oder `ulong`, je nachdem, ob der Wert in den Typ passt:

```
ulong i;
i = 10; // die Zahl 10 ist vom Typ int
i = 4294967295; // die Zahl 4294967295 ist vom Typ unit
```

Ausprobieren können Sie dies, indem Sie den Typ eines Literals ermitteln:

```
Console.WriteLine("Das Literal 4294967295 besitzt " +
   "den Typ '" + 4294967295.GetType().Name + "'");
```

Sie können einen solchen Wert aber auch einem kleineren oder größeren Datentyp zuweisen:

```
byte i;
i = 10;
long j;
j = b;
```

Wenn Sie versuchen, einem Datentyp ein zu großes oder nicht konvertierbares Literal zuzuweisen, meldet der Compiler einen Fehler:

```
byte i;
i = 1024; /* Compilerfehler "Konstantenwert '1024' kann
           * nicht nach 'byte' konvertiert werden" */
i = 10.5; /* Compilerfehler "Implizite Konvertierung
           * des Typs 'double' zu 'byte' nicht möglich"
```

Konvertierungen werden ab Seite 113 behandelt.

Sie können einem Literal einen Suffix anhängen um damit den Datentyp zu bestimmen.

Suffix	Datentyp
l	long
u	unit oder ulong, je nach Größe des Werts
ul	ulong

Tabelle 3.5: Die Suffixe für Integer-Literale

Die Groß- und Kleinschreibung spielt dabei keine Rolle.

3.3.4 Fließkommatypen

Fließkommatypen (`float`, `double und decimal`) speichern Dezimalzahlen mit einer festgelegten maximalen Anzahl an Ziffern. Die mögliche Anzahl der Dezimalstellen hängt davon ab, wie viele Ziffern vor dem Komma gespeichert sind. `float` kann z. B. nur zwei Dezimalstellen speichern, wenn fünf Ziffern vor dem Komma gespeichert sind. `float` ist deswegen normalerweise auch nicht besonders geeignet. Verwenden Sie lieber `double`, denn dieser Datentyp kann bei fünf Vorkomma-Ziffern bereits neun Dezimalstellen speichern. Bei einer Ziffer vor dem Komma kann `double` etwa fünfzehn, `float` nur etwa sieben Dezimalstellen speichern.

`decimal` ist ein spezieller Datentyp mit einer festgelegten Anzahl von Ziffern (29). Damit können Sie sich also recht schnell ausrechnen, wie viele Dezimalstellen dieser Datentyp speichern kann, wenn eine bestimmte Anzahl Ziffern vor dem Komma verwaltet wird. Diesen Datentyp können Sie verwenden, wenn Sie hochgenaue Berechnungen ausführen wollen.

Wenn Sie die Anzahl der möglichen Dezimalstellen einmal selbst ausprobieren wollen, testen Sie den folgenden Quellcode:

```
decimal d;
d = 1;
Console.WriteLine("Dezimalstellen von decimal: " +
   (d / 3));
Console.WriteLine("Dezimalstellen von double: " +
   (double)(d / 3));
Console.WriteLine("Dezimalstellen von float: " +
   (float)(d / 3));
```

Übersteigt die Anzahl der Dezimalstellen eines Ausdrucks die speicherbaren Stellen, rundet der Compiler den Wert auf die speicherbaren Dezimalstellen. Bei der folgenden Zuweisung

```
float f = 99999.125F;
```

wird der Wert auf 99999.13 gerundet. Das F steht übrigens dafür, dass es sich bei dem Literal um einen `float`-Wert handelt. Beim Runden verwendet der Compiler ein Verfahren, das dem mathematischen Verfahren ähnlich ist. Ist die Ziffer rechts von der letzten darstellbaren Ziffer eine 5, rundet der Compiler die Ziffer davor nur dann auf,

- wenn diese Ziffer < 5 ist und es sich um eine gerade Ziffer handelt oder
- wenn diese Ziffer >= 5 ist und es sich um eine ungerade Ziffer handelt.

Bei

```
float f = 99999.135F;
```

wird der Wert also auf 99999.13 abgerundet, bei

```
float f = 99999.155F;
```

wird allerdings auf 99999.16 aufgerundet.

Dieses spezielle Rundungsverfahren stellt sicher, dass beim Runden möglichst keine Ungleichheiten entstehen. Bei dem in Deutschland üblichen kaufmännischen Runden, bei dem bei einer 5 hinter der letzten im Ergebnis darzustellenden Ziffer immer aufgerundet wird, werden mehr Zahlen aufgerundet als abgerundet. Die 5 steht eben eigentlich genau in der Mitte des Zahlenstrahls.

Weitere Ziffern werden allerdings nicht berücksichtigt. Bei

```
f = 99999.13599999F;
```

wird der Wert ebenfalls auf 1.3 gerundet.

Als Literal für einen Fließkommadatentyp können Sie eine Zahl in der englischen Schreibweise verwenden. Der Compiler wertet Zahlen mit Dezimalstellen grundsätzlich als `double` aus:

```
double d = 1.234;
```

Wenn Sie einem `float`-Typ ein Literal zuweisen wollen, das einen Nachkommaanteil besitzt, müssen Sie dieses explizit als `float` kennzeichnen. Hängen Sie dazu den Suffix *f* an:

```
float f = 1.234f;
```

Integerwerte können Sie allerdings ohne Konvertierung zuweisen:

```
float f = 1;
```

Dasselbe gilt, wenn Sie einem `decimal`-Typ ein Literal zuweisen wollen. Verwenden Sie zur Konvertierung den Suffix *m*:

```
d = 1.234m; // Dezimalzahlen müssen konvertiert werden
d = 1; // Integerwerte können direkt zugewiesen werden
```

Für Dezimalzahlen können Sie auch die wissenschaftliche Schreibweise verwenden:

```
double x = 5E-2; // 0.05
```

`5E-2` steht im Beispiel für: $5 * 10^{-2}$.

Für die Festlegung des Datentyps können Sie eines der in Tabelle 3.6 dargestellten Suffixe verwenden. Die Konvertierung ist z. B. notwendig, wenn Sie einem `decimal`-Typ ein Literal zuweisen wollen, das Nachkommastellen enthält:

```
decimal d = 1.5m;
```

Suffix	Datentyp
d	double
f	float
m	decimal

Tabelle 3.6: Suffixe für die Festlegung des Datentyps eines Fließkomma-Literals

3.3.5 Über- und Unterläufe und spezielle Werte

Wenn Sie einem Typ einen zu großen Wert zuweisen, resultiert daraus ein Überlauf:

```
byte testValue = 255;
testValue += 2;
Console.WriteLine(testValue); // Ergebnis: 1
```

Der Wert in der Variablen *testValue* ist durch die Addition mit 2 übergelaufen. Wenn Sie unter Visual Studio oder bei *csc.exe* die Compiler-Voreinstellung übernehmen, resultiert ein entsprechender Wert. Im Beispiel ist das der Wert 1 (11111111_2 + 00000001_2 = 00000000_2 + 00000001_2 = 00000001_2). Wenn Sie nicht darauf achten, Ihre Datentypen ausreichend groß zu dimensionieren, kann ein Überlauf zu enormen logischen Fehlern führen, nach denen Sie u. U. tagelang (oder nächtelang) suchen.

Ein Unterlauf wird erzeugt, wenn Sie einen zu kleinen Wert zuweisen:

```
byte testValue = 0;
testValue -= 2;
Console.WriteLine(testValue); // Ergebnis: 254
```

Sie können den Compiler aber auch so einstellen, dass dieser bei einem Über- oder Unterlauf eine Ausnahme generiert. In Visual Studio stellen Sie dazu die Option AUF ARITHMETISCHEN ÜBER-/UNTERLAUF PRÜFEN ein. Sie finden diese Option in den Projekteigenschaften unter KONFIGURATIONSEIGENSCHAFTEN / ERSTELLEN. Achten Sie darauf, dass Sie die richtige Konfiguration einstellen (*Debug* oder *Release*). Sinnvoll ist die Überprüfung für die Debug-Konfiguration. Wenn Sie Ihr Programm in der Entwicklungsphase nur mit dieser Konfiguration kom-

pilieren und fleißig testen (lassen), werden Über- und Unterläufe auf jeden Fall gemeldet, sodass Sie darauf reagieren und den Fehler beseitigen können.

Im Release sollten Sie die Überprüfung u. U. nur dann einschalten, wenn Ihr Programm nicht korrekt funktioniert (um den Fehler zu finden). Die Überprüfung auf Unter- und Überlauf kostet ein wenig Rechenzeit.

Beachten Sie, dass auch ein Release nie fehlerfrei ist (wenigstens nicht bei mir ...). Wenn Sie die Über-/Unterlaufprüfung abschalten und in Ihrer Anwendung unter bestimmten Umständen Unter- oder Überläufe auftreten, wird es sehr schwierig, die Fehlerquelle zu finden. Lassen Sie diese Prüfung also auch im Release eingeschaltet, wenn Sie unsicher sind. Die minimale Rechenzeit, die die Prüfung benötigt, können moderne Systeme locker verschmerzen.

Wenn Sie den Kommandozeilencompiler verwenden, schalten Sie die Überprüfung über die Option `checked` ein:

```
csc /checked+ Quellcodedatei.cs
```

In SharpDevelop ist diese Option bereits voreingestellt. Überprüfen Sie die Projekteigenschaften auf die Option ÜBERLAUFCHECKS GENERIEREN.

Alternativ können Sie auch den `checked`-Block verwenden, um einzelne Anweisungen bei ansonsten abgeschalteter Prüfung zu überprüfen:

```
checked
{
   testValue += 2;
}
```

Alle im `checked`-Block enthaltenen Anweisungen werden nun überprüft.

Wenn Sie bei eingeschalteter Prüfung einzelne Anweisungen explizit nicht überprüfen wollen, verwenden Sie den `unchecked`-Bock:

```
unchecked
{
   testValue += 2;
}
```

Alternativ können Sie für einzelne Ausdrücke auch den `checked`- oder `unchecked`-Operator verwenden:

```
testValue = unchecked((byte)(testValue + 2));
```

Die Anweisung `(byte)` konvertiert in diesem Beispiel das `int`-Ergebnis des Ausdrucks in den Datentyp `byte`.

+0, -0, +_, -_, NaN

Die Typen `float` und `double` (nicht `decimal`!) können spezielle Werte speichern. Wenn Sie z. B. einen `double`-Typ, der eine positive Zahl speichert, durch 0 teilen, resultiert der Wert +_ (positiv unendlich):

```
double d1 = 10, d2 = 0, d3;
d3 = d1 / d2;
Console.WriteLine(d3); // Ausgabe: +unendlich
```

Teilen Sie einen negativen Wert durch 0, resultiert der Wert -_ (negativ unendlich). Wenn Sie 0 durch 0 teilen, ergibt das den Wert `NaN` (Not a Number). In einigen speziellen Fällen resultiert auch der Wert –0 oder +0. Die Regeln dazu sind ziemlich kompliziert und in einem Standard für Fließkommaoperationen beschrieben (IEEE 754). Sie finden diesen Standard unter der Adresse *grouper.ieee.org/groups/754/*. In der .NET-Framework-Hilfe finden Sie eine Tabelle, die die einzelnen Möglichkeiten auflistet. Suchen Sie nach »division operator« im Suchbereich der Hilfe und wählen Sie das Hilfethema »7.7.2 Division operator (C#)«.

Über einige statische Methoden der `double`- und der `float`-Klasse können Sie herausfinden, ob ein Typ einen dieser Werte speichert:

```
if (double.IsInfinity(d3))
   Console.WriteLine("d3 ist unendlich.");
if (double.IsNegativeInfinity(d3))
   Console.WriteLine("d3 ist negativ unendlich.");
if (double.IsPositiveInfinity(d3))
   Console.WriteLine("d3 ist positiv unendlich.");
```

```
if (double.IsNaN(d3))
   Console.WriteLine("d3 ist NaN.");
```

Die Bedeutung dieser Werte ist für die Praxis auch wohl eher gering. Vermeiden Sie einfach das Teilen durch 0, dann kann eigentlich nichts passieren.

3.3.6 Datumswerte

Datumswerte werden nicht in einem Wertetyp, sondern in einer Instanz der Klasse `DateTime` gespeichert:

```
DateTime date = new DateTime();
```

Dem Konstruktor dieser Klasse können Sie verschiedene Werte übergeben, mit denen Sie das Datum genau spezifizieren können:

```
DateTime date = new DateTime(2099,12,31) // 31.12.2099
```

Das Datum 31.12.2099 habe ich übrigens gewählt, weil ich hoffe, dass sich dieses Buch dann immer noch verkauft. Dann muss ich das Datum in den einzelnen Auflagen nicht immer wieder ändern ...

Alternativ können Sie auch eine der Klasseneigenschaften der `DateTime`-Klasse verwenden, um das aktuelle oder ein bestimmtes Datum einzustellen:

```
date = DateTime.Now;
date = DateTime.Today;
date = DateTime.Parse("31.12.2099");
```

`Now` liefert dabei das bis auf die Millisekunde genaue aktuelle Datum, `Today` das aktuelle Datum ohne Zeitangabe.

Um ein Datum auszugeben, können Sie eine der To-Methoden verwenden:

```
Console.WriteLine(date.ToLongDateString());
Console.WriteLine(date.ToShortDateString());
Console.WriteLine(date.ToLongTimeString());
Console.WriteLine(date.ToShortTimeString());
```

Ein Datumswert wird übrigens immer mit einer genauen Zeitangabe gespeichert.

Den Umgang mit Datumswerten beschreibe ich in Kapitel 5.

3.3.7 Zeichen und Zeichenketten

Für die Speicherung einzelner Zeichen verwenden Sie den Typ `char`. Zeichenketten werden im Typ `string` gespeichert. `char` ist ein Wertetyp, `string` ist ein Referenztyp (der aber ähnlich einem Wertetyp ausgewertet wird, siehe Seite 95). Zeichen werden immer im Unicode-Format gespeichert. In diesem Format wird ein Zeichen in einem zwei Byte großen Speicherbereich verwaltet. Damit sind Zeichensätze möglich, die 65535 verschiedene Zeichen speichern. Der Zeichensatz »Western Latin-1« wird in westlichen Ländern verwendet und beinhaltet alle Zeichen, die in diesen Ländern verwendet werden. Für japanische und chinesische Länder existieren andere Zeichensätze.

`char`-Literale werden in einfache Apostrophe eingeschlossen:

```
char c = 'A';
```

Alternativ können Sie auch den Unicode-Wert des Zeichens im hexadezimalen Format angeben (wenn Sie diesen kennen):

```
char c = '\u0041'; // 0x0041 = 65 = 'A'
```

Sie können auch direkt die hexadezimale Darstellung verwenden:

```
char c = '\x0041'; // 0x0041 = 65 = 'A'
```

Wenn Sie den ANSI-Code eines Zeichens kennen, setzen Sie diesen im ersten Byte ein: der ANSI-Zeichensatz ist im ersten Byte des Unicode-Zeichensatzes abgebildet.

`string`-Literale schließen Sie in Anführungszeichen ein:

```
string s = "Das ist ein String";
```

Innerhalb einer Zeichenkette können Sie auch die Unicode- oder Hexadezimal-Darstellung der Zeichen verwenden:

```
string s = "\x0041\x0042\x0043"; // "ABC"
```

Escape-Sequenzen

`char`- und `string`-Typen können so genannte Escape-Sequenzen speichern. Eine Escape-Sequenz wird immer mit einem Backslash eingeleitet. Einige Escape-Sequenzen sorgen dafür, dass Zeichen mit einer besonderen Bedeutung im String als normale Zeichen verwendet werden können. Das Anführungszeichen können Sie innerhalb eines

Strings z. B. nur über eine Escape-Sequenz verwenden, damit der Compiler dieses Zeichen nicht als Stringbegrenzer auswertet:

```
Console.WriteLine("Sein Name war \"Old Trashbarg\"");
```

Andere Escape-Sequenzen stehen für spezifische Zeichen, wie z. B. einen Zeilenumbruch:

```
Console.WriteLine("Das ist Zeile 1\nDas ist Zeile 2");
```

Tabelle 3.7 listet die Escape-Sequenzen von C# auf.

Escape-Sequenz	Bedeutung
\'	Apostroph
\"	Anführungszeichen
\\	Backslash
\o	Null-Zeichen
\a	Warnung (alert). Erzeugt einen Piepston.
\b	Backspace
\f	Seitenvorschub (Form Feed)
\n	Zeilenumbruch (New Line)
\r	Wagenrücklauf (Carriage Return)
\t	Horizontaler Tabulator
\v	Vertikaler Tabulator

Tabelle 3.7: Die C#-Escape-Sequenzen

Sehr häufig benötigen Sie in Strings einen Zeilenumbruch. Oft reicht dafür aber nicht allein die Escape-Sequenz \n aus, weil viele Ausgabegeräte zusätzlich dazu einen Wagenrücklauf benötigen (das klassische Carriage Return /Line Feed). Verwenden Sie dann eine Kombination: \r\n. Achten Sie auf die Reihenfolge: Erst das Carriage Return, dann das Line Feed.

Wortwörtliche Stringliterale

In normalen Stringliteralen werden Escape-Sequenzen ausgewertet. In einigen Fällen kann das zum Problem werden. Wenn Sie z. B. einen

Dateipfad angeben, wollen Sie die Backslashs nicht als Escape-Zeichen verwenden. Häufig meckert schon der Compiler:

```
string filename = "c:\files\books\nitty-gritty\C#.doc";
```

In diesem Fall meldet der Compiler eine »Nicht erkannte Escape-Folge". Enthält die Zeichenfolge zufällig nur gültige Escape-Sequenzen, ist ein Programmfehler vordefiniert:

```
string filename = "c:\files\nitty-gritty\release.doc";
Console.WriteLine(filename);
```

Bild 3.4: Ausgabe des fehlerhaften Strings

Sie können die Escape-Sequenzen über doppelte Backslashs deaktivieren:

```
string filename =
   "c:\\files\\nitty-gritty\\release.doc";
```

Einfacher und besser ist, stattdessen »wortwörtliche« Stringliterale (Verbatim String Literals) zu verwenden. Diese Literale, die keine Escape-Sequenzen auswerten, leiten Sie über ein @ ein:

```
string filename = @"c:\files\nitty-gritty\release.doc";
```

Wortwörtliche Stringliterale können sogar in mehreren Zeilen umbrochen werden, ohne dass Sie diese in jeder Zeile abschließen und mit + addieren müssen:

```
s = @"Das ist ein
 wortwörtliches Stringliteral, das einfach
 umbrochen wurde";
```

Beachten Sie, dass nun alle Zeilenvorschübe und Leerzeichen mit in den String übernommen werden. Der String hat also genau dasselbe Format, wie es im Quellcode angegeben ist.

Strings

Strings werden automatisch dynamisch verwaltet. Die Größe des Strings ist lediglich durch den verfügbaren Speicher begrenzt. Wenn Sie einen String neu zuweisen oder verändern, erzeugt das Programm immer einen neuen Speicherbereich:

```
String s = "Das ist ein";
s = s + "Test"; // erzeugt einen neuen Speicherbereich
```

In Stringausdrücken ruft der Compiler implizit die `ToString`-Methode anderer Typen auf, wenn Sie das nicht machen. Deshalb können Sie beispielsweise einen `int`-Typ mit einem String ganz einfach addieren:

```
int i = 42;
string s = i + " ist eine gute Zahl";
```

Wenn Sie viel mit einem String arbeiten, ist das nicht sehr effizient. Dann können Sie alternativ eine Instanz der `StringBuilder`-Klasse verwenden, die das Umkopieren vermeidet (siehe Kapitel 5).

Die `string`-Klasse besitzt eine große Anzahl an Methoden zur Arbeit mit Zeichenketten. In Kapitel 5 beschreibe ich, wie Sie diese nutzen.

3.3.8 Der Typ object

Der Typ `object` ist ein besonderer Typ. Diesem Typ können Sie jeden anderen Typ zuweisen:

```
object o;
o = 10;
o = "Hallo";
```

Viele Methoden besitzen Argumente vom Typ `object`. Das ist auch der Haupt-Anwendungsbereich dieses Typs. `object`-Argumenten können Sie beliebige Werte übergeben. Die `WriteLine`-Methode der `Console`-Klasse ist ein Beispiel dafür. Methoden, die den übergebenen Wert als String auswerten, nutzen dabei die `ToString`-Methode, die jeder Typ von `object` geerbt hat.

Wenn Sie einen `object`-Typ in einem nummerischen Ausdruck verwenden, müssen Sie diesen in den entsprechenden Typ konvertieren:

```
object o = 10;
int i = (int)o * 10;
```

In Stringausdrücken ist das allerdings nicht notwendig, da der Compiler dann automatisch die `ToString`-Methode des `object`-Typen aufruft:

```
object o = "42";
string s = o + " ist eine gute Zahl";
```

`object` merkt sich den gespeicherten Typ. Über die `GetType`-Methode können Sie den Typ ermitteln:

```
Console.WriteLine(o.GetType().Name);
```

Beachten Sie, dass `GetType` ein Objekt der Klasse `Type` zurückgibt, über das Sie noch wesentlich mehr Informationen über den Typ erhalten können, als nur dessen Name. Die Eigenschaft `IsArray` ist z. B. `true`, wenn der Typ ein Array ist.

Verwenden Sie den Typ `object` *idealerweise nur in Ausnahmefällen. Die Typsicherheit von C# geht mit diesem Typ verloren. Wenn Sie* `object`*-Variablen verwenden, wissen Sie in größeren Programmen nie genau, welchen Datentyp diese tatsächlich speichern. Logische Programmfehler und Ausnahmefehler sind damit vorprogrammiert.*

Boxing und Unboxing

Der Datentyp `object` ist ein Referenztyp. Boxing und Unboxing sind Techniken, die im Hintergrund verwendet werden und die ermöglichen, dass der Typ `object` nicht nur Referenztypen, sondern auch Wertetypen verwalten kann. Wenn ein Wertetyp an einen `object`-Typ zugewiesen wird, wird implizit Boxing verwendet:

```
int i = 10;
object o = i; // i wird geboxt
```

Boxing ist eine implizite Konvertierung eines Wertetypen in den Typ `object`. Da `object` ein Referenztyp ist, werden die verwalteten Werte folglich auf dem Heap gespeichert. Wenn Sie einen Wertetypen an eine `object`-Variable zuweisen oder an ein `object`-Argument übergeben, erzeugt der Compiler eine »Box« auf dem Heap und kopiert den Wert des Wertetypen in diese Box. Die Box simuliert einen Referenztypen und kann deshalb über eine `object`-Variable verwendet werden.

Unboxing bedeutet, dass ein `object`-Typ in einen Wertetyp konvertiert wird (was immer explizit geschehen muss):

```
object o = 10;
int i = (int)o; // der Wert von o wird in den
                // Wertetyp »entboxt«
```

3.3.9 Konvertierungen

Wie Sie ja bereits wissen, ist C# eine typsichere Sprache. Sie können einem Typ nur einen passenden anderen Typ zuweisen, dessen Maximal- bzw. Minimalwert ohne Kürzungen in den anderen Typ passt. Einer `int`-Variable können Sie z. B. ohne Probleme einen `byte`-Wert zuweisen:

```
byte b = 255;
int i = b;
```

Der Compiler konvertiert den Datentyp dann implizit. Wenn der Grundtyp aber nicht passt oder der Wert nicht in den zugewiesenen Typ passen würde, erzeugt der Compiler einen Fehler:

```
string s = "255";
int i = s; // Fehler
float f = 1.234f;
i = f; // Fehler
```

Literale können Sie über die Datentypszeichen auf einen bestimmten Datentyp festlegen. Das Literal 1.234 habe ich im Beispiel über das Zeichen f zu einem `float`-Typ konvertiert. Andere Typen müssen Sie über einen Cast oder mit Hilfe der Methoden der `Convert`-Klasse explizit konvertieren.

Explizite und implizite Konvertierungen bei Typen

Jeder Typ kann spezielle Operatoren für implizite und explizite Konvertierungen enthalten. Solch ein Operator ist so etwas wie eine Methode, die allerdings automatisch aufgerufen wird, wenn der Typ konvertiert werden soll. Konvertierungs-Operatoren können für die verschiedensten anderen Typen definiert sein. Wenn Sie eine eigene Klasse oder Struktur erzeugen, können Sie beliebig viele Konvertierungs-Operatoren erzeugen, die dann die implizite oder explizite

Konvertierung der verschiedenen Typen erlauben. Kapitel 4 zeigt, wie das programmiert wird. Ein `int`-Typ lässt z. B. (u. a.) die implizite Konvertierung eines `byte`- und eines `short`-Typs und die explizite Konvertierung eines `float`- und eines `double`-Typs zu:

```
int i = 0; byte b = 10; float f = 1.2345f;
i = b; // Implizite Konvertierung
i = (int)f; // Explizite Konvertierung. Die
            // Dezimalstellen gehen verloren.
```

Bei impliziten Konvertierungen geht nichts verloren, bei expliziten Konvertierungen kann es sein, dass Informationen verloren gehen. Explizite Konvertierungen nehmen Sie über einen Cast vor. Alle Datentypen, die denselben Grundtyp besitzen, ermöglichen einen Cast.

Cast

Wenn ein Typ einen speziellen Operator besitzt, der die explizite Konvertierung eines anderen Typs erlaubt, können Sie diesen über einen Cast konvertieren. Dazu setzen Sie den Typ in Klammern vor den zu konvertierenden Ausdruck. Der `float`-Typ erlaubt z. B. die Konvertierung eines `double`-Typs.

```
double d = 1.2345678901234567890;
float f = (float)d;
```

Beim Konvertieren kann es sein, dass Informationen verloren gehen. Die `float`-Variable des Beispiels speichert nach der Ausführung z. B. den Wert 1,234568. Wie in C# üblich, wird dabei die letzte Ziffer annähernd mathematisch gerundet (siehe bei »Fließkommatypen« ab Seite 102).

Wenn Sie Ausdrücke casten wollen, müssen Sie den Ausdruck klammern. Der cast-Operator gilt immer nur für den Operanden rechts von ihm:

```
f = (float)(1.234567890 * 1.234567890);
```

Konvertierungen über die Convert-Klasse

Datentypen, die nicht denselben Grundtyp besitzen, können nicht einfach über einen Cast konvertiert werden. So können Sie einen String z. B. nicht in einen `int`-Wert casten:

```
string s = "10";
int i = (int)s; // Fehler
```

Solche Konvertierungen müssen Sie über die Klassenmethoden der `Convert`-Klasse vornehmen. Der Grund dafür liegt darin, dass Konvertierungen zwischen grundverschiedenen Datentypen auch fehlschlagen können. Die Methoden der `Convert`-Klasse erzeugen in diesem Fall eine Ausnahme.

Einen String konvertieren Sie z. B. über die `ToInt32`-Methode in einen Integer:

```
s = "10";
i = Convert.ToInt32(s);
```

Schlägt die Konvertierung fehl, erzeugt die Methode eine Ausnahme vom Typ `FormatException`:

```
s = "10a";
i = Convert.ToInt32(s); // Ausnahme
```

Diese Ausnahme können Sie abfangen, wie ich es noch in Kapitel 6 zeige.

Tabelle 3.8 zeigt die wichtigsten Klassenmethoden der `Convert`-Klasse. Diese Methoden liegen in mehreren Varianten vor, die die unterschiedlichen Datentypen übergeben bekommen. Ich beschreibe nicht alle diese Varianten und erläutere deswegen die Grundlagen: Die Varianten, die keinen `string` übergeben bekommen, besitzen nur ein Argument. Der `ToInt32`-Methode können Sie z. B. im ersten Argument (u. a.) einen `float`-, `short`- oder `double`-Wert übergeben. Die Variante, die einen String übergeben bekommt, kann zusätzlich noch mit einem zweiten Argument aufgerufen werden, in dem Sie einen Format-Provider übergeben, der die länderspezifische Formatierung des Strings festlegt. Übergeben Sie diesen nicht, wird die Systemeinstellung berücksichtigt.

Methode	konvertiert in
`ToBoolean(...)`	`bool`
`ToByte(...)`	`byte`
`ToChar(...)`	`char`

Methode	konvertiert in
ToDateTime(...)	DateTime
ToDecimal(...)	decimal
ToDouble(...)	double
ToInt16(...)	short
ToInt32(...)	int
ToInt64(...)	long
ToSByte(...)	sbyte
ToSingle(...)	float
ToString(...)	string
ToUInt16(...)	ushort
ToUInt32(...)	uint
ToUInt64(...)	ulong

Tabelle 3.8: Die wichtigsten Klassenmethoden der Klasse Convert

Für Konvertierungen in Strings können Sie übrigens auch die `ToString`-Methode verwenden, die alle Typen besitzen.

3.3.10 Aufzählungen (Enums)

Aufzählungen sind Datentypen, die aus mehreren benannten Zahlkonstanten bestehen. Eine Aufzählungen wird nach dem folgenden Schema deklariert:

```
[Attribute] [Modifizierer] enum Name
   [: Basistyp] {Konstantenliste};
```

In der Liste geben Sie einen oder mehrere Bezeichner an:

```
private enum Direction {North, South, East, West}
```

Wenn Sie keinen Datentyp angeben, besitzt die Auflistung den Typ `int`. Sie können den Datentyp aber auch definieren:

```
private enum Direction: byte {North, South, East, West}
```

Verwenden können Sie die Integer-Typen (`byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` und `ulong`). Diesen neuen Typ können Sie nun überall

da einsetzen, wo er gültig ist. Eine Methode kann z. B. ein Argument dieses Typs besitzen:

```
static void Move(Direction direction)
{
   if (direction = Direction.North)
      ...
}
```

Beim Aufruf der Methode muss nun ein passender Typ übergeben werden:

```
Move(Direction.North);
```

Wenn Sie bei der Deklaration der Auflistung keinen Wert für die einzelnen Konstanten angeben, erhalten diese einen bei o beginnenden Wert. Sie können den Wert allerdings auch festlegen:

```
private enum Direction: byte
   {North = 1, South = 2, East = 4, West = 8}
```

Wenn Sie die Werte wie im Beispiel so definieren, dass diese die einzelnen Bits der Typs repräsentieren, können Sie die Auflistung bitweise einsetzen:

```
Move(Direction.North | Direction.West);
```

und natürlich auswerten:

```
static void Move(Direction direction)
{
   if ((direction & Direction.North) > 0)
      Console.WriteLine("Going North");
   if ((direction & Direction.South) > 0)
      Console.WriteLine("Going South");
   if ((direction & Direction.West) > 0)
      Console.WriteLine("Going West");
   if ((direction & Direction.East) > 0)
      Console.WriteLine("Going East");
}
```

Auflistungen können natürlich konvertiert werden, was aber normalerweise explizit erfolgen muss:

```
Direction d = Direction.North;
byte i = (byte)d;
...
i = 255;
d = (Direction)i;
```

Tipps

Häufig ist es notwendig den Namen eines oder mehrerer Felder einer Auflistung zu ermitteln, wenn Sie nur den Wert des Feldes kennen. Sie können dann einfach die Methode `Enum.GetName(Type` *enumType*`, object` *value*`)` verwenden. Das folgende Beispiel ermittelt den Namen der Konstante mit dem Wert 2:

```
Console.WriteLine(Enum.GetName(typeof(Direction),2));
```

Über `GetNames` können Sie auch alle Konstanten durchgehen:

```
foreach (string name in Enum.GetNames(typeof(Direction)))
   Console.WriteLine(name);
```

3.4 Variablen, Konstanten und Arrays

In Variablen und Konstanten können Sie Werte speichern und Objekte referenzieren. Eine Variable oder eine Konstante besitzt wie alle Typen einen Datentyp. Variablen können im Programm verändert werden, der Wert einer Konstante ist unveränderlich. C# unterscheidet die üblichen Bereiche für die Lebensdauer und die Gültigkeit. Variablen und Konstanten können Sie innerhalb einer Methode deklarieren. Dann gilt die Variable nur innerhalb der Methode und lebt nur so lange, wie die Methode ausgeführt wird. Eine andere Möglichkeit der Deklaration ist innerhalb einer Klasse. Dabei unterscheidet C# zwischen normalen und statischen Variablen (die dann allerdings als Eigenschaft bezeichnet werden). Ich beschreibe hier nur die Deklaration innerhalb einer Methode, weil Kapitel 4 ausführlich auf normale und statische Eigenschaften eingeht.

3.4.1 Deklaration von Variablen und Konstanten

Bei der Deklaration einer Variablen geben Sie optionale Attribute, einen optionalen Modifizierer, den Datentyp und den Bezeichner an. Eine Variable können Sie direkt bei der Deklaration initialisieren:

```
[Attribute] [Modifizierer] Typ Name [= Ausdruck];
```

Über Attribute können Sie zusätzliche Informationen zu einer Variablen definieren, die entweder vom Compiler oder von externen Programmen ausgewertet werden können. Attribute werden ab Seite 149 behandelt.

Der Modifizierer wird bei Eigenschaften verwendet, die Sie innerhalb von Klassen deklarieren (siehe Kapitel 4) und legt den Gültigkeitsbereich der Eigenschaft fest.

Die Regeln für den Bezeichner sind die üblichen: Verwenden Sie nur Buchstaben, Zahlen und den Unterstrich und beginnen Sie den Bezeichner mit einem Buchstaben oder dem Unterstrich.

Innerhalb einer Methode wird kein Modifizierer verwendet. Die folgende Anweisung deklariert eine `int`-Variable und initialisiert deren Wert auf 0:

```
int i = 0;
```

Sie können in einer Anweisung gleich mehrere Variablen deklarieren. Trennen Sie die einzelnen Variablen durch Kommata:

```
int i = 0, j = 0, k = 0;
```

Die der ersten Deklaration folgenden Deklarationen erhalten denselben Datentyp.

In C# gelten Variablen blockweise. Wenn Sie eine Variable in einem Block deklarieren, können Sie außerhalb des Blocks nicht auf die Variable zugreifen:

```
/* Deklaration innerhalb einer Abfrage */
if (i == 0)
{
   string s = "i ist 0";
}
Console.WriteLine(s); // dieser Zugriff ist nicht
                      // möglich
```

Daneben können Sie innerhalb einer Methode keine Variable erneut deklarieren, die in einem untergeordneten Block bereits deklariert wurde:

```
/* Deklaration innerhalb einer Abfrage */
if (i == 0)
{
   string s = "i ist 0";
   Console.WriteLine(s);
}
string s = ""; // diese Deklaration ist nicht
               // möglich
```

Sie können aber innerhalb der Klasse Eigenschaften deklarieren, die denselben Namen tragen wie Variablen in einer Methode.

Deklaration von Konstanten

Konstanten werden ähnlich wie Variablen deklariert:

`[`*`Attribute`*`] [`*`Modifizierer`*`] const `*`Typ Name = Ausdruck`*`;`

Bei einer Konstantendeklaration müssen Sie direkt einen Wert zuweisen. Die Modifizierer gelten wieder für die Deklaration auf Klassenebene. Den Wert einer Konstanten können Sie im Programm nicht mehr ändern.

Setzen Sie überall da, wo Sie mit festen Werten arbeiten, die sich eventuell einmal ändern können, grundsätzlich besser Konstanten ein. Wenn ein solcher Wert später geändert werden muss, können Sie einfach die Konstante anpassen. Auch die Umsetzung einer Konstanten in eine Variable oder Eigenschaft ist einfach. Dann können Sie den Wert z. B. aus einer Konfigurationsdatei auslesen um dem Anwender die Konfiguration des Programms zu ermöglichen.

3.4.2 Arrays

Arrays erlauben das zusammenhängende Speichern mehrerer gleichartiger Informationen. Arrays verhalten sich wie eine Liste einzelner Variablen, auf die Sie über einen Namen und einen Index zugreifen können.

Das .NET-Framework stellt neben Arrays noch viele weitere Klassen zur Verfügung, über die Sie Daten zusammenhängend speichern können. Diese Klassen, die eine wesentlich flexiblere Arbeit mit den Daten ermöglichen, stelle ich in Kapitel 5 vor.

C#-Arrays sind komplexe Objekte, die einige Operationen auf den gespeicherten Daten erlauben. So können Sie ein Array beispielsweise über die (statische) `Sort`-Methode sortieren und über die `IndexOf`- oder `BinarySearch`-Methode durchsuchen.

Arrays erzeugen

In C# sind Arrays Objekte, die Sie vor der Benutzung erzeugen müssen:

```
Typ[] Name = new Typ[AnzahlElemente];
```

Der Datentyp der im Array gespeicherten Elemente kann ein beliebiger Typ sein.

Ein Array, das drei `int`-Werte speichert, deklarieren Sie beispielsweise so:

```
int[] intArray = new int[3];
```

Weil Arrays Objekte sind, können Sie der Arrayvariablen auch nachträglich größere oder kleinere Arrays zuweisen:

```
intArray = new int[5];
```

Die alten Inhalte gehen dabei natürlich verloren.

Um auf die einzelnen Elemente eines Arrays zuzugreifen, geben Sie den Index des Elements im Array in eckigen Klammern an. C#-Arrays beginnen immer mit dem Index 0:

```
int i;
intArray[0] = 10; // erstes Element beschreiben
i = intArray[0]; // erstes Element auslesen
```

Wenn Sie auf alle Elemente eines Arrays sequenziell zugreifen wollen, können Sie dies in einer `for`-Schleife machen. Über die Eigenschaft `Length` erhalten Sie die Anzahl der im Array gespeicherten Elemente:

```
for (int i=0; i<intArray.Length; i++)
{
   Console.Write(intArray[i] + " ");
}
```

Beachten Sie, dass `Length` *die Gesamtanzahl der gespeicherten Elemente zurückgibt. Handelt es sich um ein mehrdimensionales Array, können Sie* `Length` *nicht verwenden. Setzen Sie dann die* `GetLength`*-Methode ein, der Sie die Dimension übergeben können, deren Elementzahl Sie ermitteln wollen.*

Alternativ können Sie die `foreach`-Schleife verwenden, die im Vergleich zur `for`-Schleife schneller ausgeführt wird:

```
foreach (int i in intArray)
{
   Console.Write(i + " ");
}
```

Innerhalb dieser Schleife haben Sie über die Variable, die Sie in den Klammern angeben, Zugriff auf die einzelnen Elemente. Diese Variable ist allerdings schreibgeschützt, Sie können also nichts hinein schreiben. Handelt es sich bei den gespeicherten Elementen um Referenztypen, können Sie diese natürlich auch über die Variable bearbeiten.

Mehrdimensionale Arrays erzeugen Sie, indem Sie die Dimensionen bei der Deklaration durch Kommata angeben und bei der Erzeugung des Arrays für jede Dimensionen deren Obergrenze festlegen. Ein zweidimensionales `string`-Array mit zwei Elementen in der ersten und drei Elementen in der zweiten Dimension erzeugen Sie z. B. so:

```
string[,] persons = new string[2,3];
```

Der Zugriff auf ein solches mehrdimensionale Array erfolgt wie bei einem eindimensionalen, nur dass Sie eben die Indizes der einzelnen Dimensionen angeben müssen. Das folgenden Listing speichert die Daten von Personen im erzeugten Array:

```
persons[0,0] = "Fred-Bogus";
persons[0,1] = "Trumper";
persons[0,2] = "New York";
```

```
persons[1,0] = "Merril";
persons[1,1] = "Overturf";
persons[1,2] = "New York";
```

Zweidimensionale Arrays werden heute eigentlich kaum noch benötigt. Für die Speicherung von strukturierten Daten sind Objekte viel besser geeignet. So können Sie ein eindimensionales Array verwenden, um zusammenhängende, strukturierte Daten in Objekten zu speichern. In Kapitel 4 erfahren Sie, wie Sie Klassen erzeugen, in Kapitel 5, wie Sie Daten idealerweise im Programm verwalten.

Arrays initialisieren

Speichert ein Array Wertetypen, werden diese bei der Erzeugung des Arrays direkt initialisiert. Die `Array`-Klasse ruft dazu den Standardkonstruktor des Typs auf, der den Wert in der Regel auf einen Leerwert setzt. Sie können diese Initialisierung jederzeit wiederholen, indem Sie die `Initialize`-Methode aufrufen.

Sie können ein Arrays aber auch direkt bei der Erzeugung mit definierten Werten initialisieren, indem Sie die zu speichernden Werte in geschweiften Klammern, getrennt durch Kommata angeben. Dann dürfen Sie allerdings keine Index-Grenzen angeben:

```
string[] bohemians = new String[] {"Löwenherz",
    "Myoko", "Ragnarök", "Prediger", "Z.Z. Top"};
```

Wollen Sie ein mehrdimensionales Array initialisieren, geben Sie die Werte der einzelnen Dimensionen wieder in geschweiften Klammern an:

```
string[,] persons = new string[,] {
   {"Fred-Bogus","Trumper", "New York"},
   {"Merril", "Overturf", "New York"}};
```

Arrays im Array

C# unterstützt auch Arrays von Arrays (Jagged Arrays). Eigentlich ist das vollkommen logisch, denn Arrays können beliebige Datentypen speichern, eben auch wieder Arrays. Diese Tatsache verdient trotzdem ein wenig Beachtung, denn die Verwendung und Initialisierung eines solchen Arrays ist etwas anders, als bei normalen.

Zunächst können Sie bei der Erzeugung eines mehrdimensionalen Arrays für jedes Element der ersten Dimension in der zweiten Dimension unterschiedlich viele Elemente angeben. Anders ausgedrückt kann jede »Zeile« in einem solchen Array unterschiedlich viele »Spalten« besitzen:

```
int[][] jaggedIntArray = new int[2][];

/* Für jedes Element der ersten Dimension
 * eine unterschiedlich große Anzahl
 * Elemente der zweiten Dimension festlegen */
jaggedIntArray[0] = new int[2];
jaggedIntArray[1] = new int[3];

/* Nun können Sie auf die unterschiedlichen
 * Elemente zugreifen */
jaggedIntArray[0][0] = 10;
jaggedIntArray[0][1] = 11;
jaggedIntArray[1][0] = 20;
jaggedIntArray[1][1] = 21;
jaggedIntArray[1][2] = 22;
```

Arrays im Array sollen in bestimmten Programmsituationen schneller sein als normale mehrdimensionale Arrays.

Wollen Sie ein Jagged Array direkt bei der Erzeugung initialisieren, müssen Sie in den geschweiften Klammern neue Arrays erzeugen:

```
string[][] persons = new string[][] {
   new string[] {"Fred-Bogus", "Trumper", "New York"},
   new string[] {"Merril", "Overturf", "New York"}};
```

Der Zugriff auf ein Array von Arrays erfolgt, indem Sie die Indizes in separaten eckigen Klammern angeben:

```
Console.WriteLine(persons[0][0] + " " +
   persons[0][1] + " wohnt in " +
   persons[0][2]);
```

Zuweisungen von Arrays

Beim Zuweisen von Arrays müssen Sie beachten, dass Arrays Referenztypen sind. Wenn Sie ein Array einem anderen zuweisen, wird lediglich die Referenz kopiert:

```
int[] intArray1 = new int[] {1,2,3};
int[] intArray2 = new int[] {10,20,30};
intArray1 = intArray2;
```

In diesem Beispiel zeigen nun beide Variablen auf dasselbe Array. Das Array das die Variable *intArray1* referenziert hat, steht nicht mehr zur Verfügung, und wird vom Garbage Collector entsorgt. Wenn Sie das Array nun über eine der Variablen verändern, ist die Veränderung natürlich auch über die andere Variable sichtbar.

Wenn Sie Arrays wirklich kopieren wollen, müssen Sie die `Clone`-Methode verwenden. Diese Methode gibt allerdings einen `object`-Typ zurück, den Sie in den korrekten Array-Typ casten müssen:

```
int[] intArray1 = new int[] {1,2,3};
int[] intArray2;
intArray2 = (int[])intArray1.Clone();
```

Da nun beide Variablen auf verschiedene Arrays verweisen, betrifft die Veränderung eines Arrays nicht mehr das andere.

Methoden und Eigenschaften eines Arrays

Alle C#-Arrays sind von der Klasse `Array` abgeleitet und besitzen deswegen die Eigenschaften und Methoden dieser Klasse. Wie viele andere .NET-Klassen besitzt auch die `Array`-Klasse Instanz- und Klassenelemente. Die wichtigsten Instanzeigenschaften und -methoden zeigt Tabelle 3.9.

Eigenschaft/ Methode	Beschreibung
`Array Clone()`	erzeugt eine Kopie eines Arrays

`void CopyTo(`
`Array` *`array`*`,`
`int` *`index`*`)`
	kopiert den Inhalt eines Arrays in ein anderes Array. In *`index`* geben Sie den Index des Zielarrays an, ab dem das Kopieren beginnen soll.

`int GetLength(` `int` *`dimension`*`)`	ermittelt die Anzahl der Elemente in einer gegebenen Dimension
`void Initialize()`	Über diese Methode können Sie die einzelnen Elemente eines Arrays auf einen Standardwert (in der Regel ein Leerwert) initialisieren. `Initialize` ruft dazu einfach den Standardkonstruktor der gespeicherten Typen auf.

Eigenschaft/ Methode	Beschreibung
`Length`	Diese Eigenschaft gibt die Gesamtanzahl der gespeicherten Elemente zurück .
`Rank`	Über diese Eigenschaft können Sie die Gesamtanzahl der Dimensionen ermitteln.

Tabelle 3.9: Die wichtigsten Instanzeigenschaften und -methoden der Array-Klasse

Die `Array`*-Klasse besitzt die Methoden* `GetLowerBound` *und* `GetUpperBound`*, über die Sie die Grenzen einer Dimension ermitteln können. Besonders* `GetLowerBound` *ist in C# ziemlich sinnlos, weil C#-Arrays immer bei 0 beginnen. Das muss in den verschiedenen .NET-Sprachen aber nicht immer der Fall sein. Einige Sprachen erlauben eventuell auch, dass Arrays mit einem anderen Index beginnen, da evtl. sogar dynamisch eingestellt werden kann. Diese Sprachen können dann über diese Methoden die Grenzen ermitteln.*

Wie viele anderer .NET-Klassen, besitzt auch die `Array`-Klasse Klassenmethoden. Die wichtigsten beschreibt Tabelle 3.10.

Eigenschaft/Methode	Beschreibung

`int BinarySearch(`
`Array` *`array`*`,`
`object` *`value`*
`[, IComparer` *`comparer`*`])`
`int BinarySearch(`
`Array` *`array`*`,`
`int` *`index`*`,`
`int` *`length`*`,`
`object` *`value`*
`[, IComparer` *`comparer`*`])`
	Über diese Methode können Sie einen Wert im Array über die schnelle binäre Suche suchen. Das Array muss dazu sortiert sein. Wenn Sie im dritten Argument einen Index angeben, durchsucht die Methode ab diesem Index die in *`length`* angegebene Anzahl Elemente. Geben Sie keinen Index an, wird das ganze Array durchsucht. Im letzten Argument können Sie ein Objekt übergeben, das die Schnittstelle `IComparer` implementiert, über deren `Compare`-Methode der Vergleich der Elemente erfolgt. So können Sie benutzerdefinierte Suchen programmieren. Die Standarddatentypen implementieren diese Schnitt-

Eigenschaft/Methode	Beschreibung
	stelle bereits, weswegen Sie diese auch ohne `IComparer`-Objekt durchsuchen können. Wie Sie eigene Klassen programmieren, die die binäre Suche ermöglichen, zeige ich in Kapitel 4.

`int IndexOf(`
` Array array,`
` object value`
` [, int startIndex]`
` [, int count]`
	Über `IndexOf` können Sie ein Array sequenziell (und damit langsam) durchsuchen.

`int IndexOf(`
` Array array,`
` object value`
` [, int startIndex]`
` [, int count]`
	`LastIndexOf` durchsucht ein Array sequenziell, ermittelt aber den Index des letzten gespeicherten Werts, der zum Suchwert identisch ist.

`Array Sort(Array array;`
` [, int index,`
` int length]`
` [, IComparer comparer])`
 	
`Array Sort(Array array;`
` , IComparer comparer)`
 	
`Array Sort(`
` Array keys,`
` Array items`
` [, int index,`
` int length]`
` [, IComparer comparer])`
 	
`Array Sort(`
` Array keys,`
` Array items`
` , IComparer comparer)`
	Über diese Methode können Sie ein Array ganz oder teilweise sortieren. Die gespeicherten Typen müssen dazu die `IComparer`-Schnittstelle implementieren, was bei den meisten Typen bereits der Fall ist. Für eine benutzerdefinierte Sortierung können Sie ein separates Objekt übergeben, dass diese Schnittstelle implementiert. In Kapitel 4 zeige ich, wie Sie eigene Klassen erzeugen, die sortierbar sind.

Tabelle 3.10: Die wichtigsten Klassenmethoden der Array-Klasse

3.4.3 Namensrichtlinien

Bei der Programmierung macht es immer Sinn, sich an gewisse Richtlinien bei der Vergabe von Bezeichnern zu halten. Damit machen Sie es anderen Programmierern leichter, Ihren Quelltext zu verstehen. In verschiedenen Programmiersprachen existieren unterschiedliche Notationen für die Benennung. C++-Programmierer verwenden z. B. meist die so genannte »Ungarische Notation«, Visual Basic-Programmierer wenden die »Reddik-Konvention« an. Für C# beschreibt Microsoft in der C#-Dokumentation eine Richtlinie für die Benennung von Bezeichnern, die recht übersichtlich ist. Diskussionen in verschiedenen Newsgroups zeigen, dass sich diese Richtlinien durchsetzen.

Die Grundlage der Benennung von Bezeichnern ist das so genannte PascalCasing und das camelCasing. Beim PascalCasing beginnt der Bezeichner mit einem Großbuchstaben und wird dann klein weitergeschrieben. Besteht der Bezeichner aus mehreren Worten, wird jedes Wort wieder mit einem Großbuchstaben begonnen:

```
public int LeftMargin;
```

Das camelCasing ist ähnlich, nur dass der Bezeichner mit einem Kleinbuchstaben begonnen wird:

```
int leftMargin;
```

Das PascalCasing wird hauptsächlich bei öffentlichen Elementen verwendet, das camelCasing bei privaten oder lokalen Elementen. Tabelle 3.11 fasst die Richtlinien zusammen.

Element	Namensrichtlinie
Klasse	▪ PascalCasing
Schnittstelle	▪ PascalCasing ▪ »I« als Präfix
Aufzählungen (Enums)	▪ PascalCasing (für den Namen der Aufzählung und die Werte)
Eigenschaften	▪ Bennennung mit Substantiven (Hauptwörtern) oder Substantiv-Phrasen (z. B. *Color*, *FirstName*) ▪ PascalCasing für öffentliche Eigenschaften ▪ camelCasing für private und geschützte Eigenschaften

Element	Namensrichtlinie
Methoden	▮ Bennennung mit Verben oder Verb-Phrasen (z. B. *Remove*, *RemoveAll*) ▮ PascalCasing
Ereignisse	▮ Benennen Sie Ereignisbehandlungsmethoden mit dem »EventHandler«-Suffix (z. B. *MouseEventHandler*) ▮ Verwenden Sie die zwei Argumente *sender* und *e* ▮ Bennennen Sie Ereignisargument-Klassen mit dem Suffix »EventArgs« (z. B. *MouseEventArgs*) ▮ PascalCasing
Argumente	▮ Verwenden Sie aussagekräftige Namen ▮ camelCasing

Tabelle 3.11: Namensrichtlinien für C#-Programme

Präfixe werden (von fast allen C#-Programmierern, wie Recherchen in Newsgroups zeigten) grundsätzlich nicht verwendet. Die C#-Konvention weicht damit erheblich von anderen Konventionen ab, die eine Präfix vor den eigentlchen Namen setzen, der den Datentyp kennzeichnet. Diese in C++ als ungarische Notation und in Visual Basic als Reddik-Konvention bezeichnete Konvention bringt in der Praxis aber unter C# nicht allzu viele Vorteile. C# ist typsicher, Sie müssen also nicht wissen, welchen Typ ein Bezeichner hat: Der Compiler meldet bei falschen Zuweisungen einen Fehler und Sie müssen explizit konvertieren, wenn der Typ nicht implizit konvertiert werden kann.

3.5 Ausdrücke und Operatoren

3.5.1 Arithmetische Ausdrücke und Operatoren

Arithmetische Ausdrücke ergeben einen Wert (eine Zahl, ein Datum, eine Zeichenkette), der in weiteren Ausdrücken oder in Zuweisungen verwendet werden kann. Der Ausdruck

```
1 + 1
```

ergibt z. B. den Wert 2 (wenn ich richtig gerechnet habe ...).

Arithmetische Ausdrücke verwenden die in Tabelle 3.12 beschriebenen Operatoren, von denen einige unär sind (d. h. nur einen Operanden besitzen) und einige binär (d. h. zwei Operanden besitzen).

Operator	Bedeutung
()	Klammern; Verschieben der Rechenpriorität
+	Addition zweier Operanden
++	Addition eines Operanden mit 1
-	Subtraktion zweier Operanden
--	Subtraktion von 1 von einem Operanden
*	Multiplikation
/	Division
%	Modulo-Division

Tabelle 3.12: Die arithmetischen Operatoren

Über den Operator + können Sie auch Strings verketten:

```
String firstName = "Donald";
String surName = "Duck";
Console.WriteLine(firstName + " " + surName);
```

Die beiden Operanden ++ und – arbeiten unär. Die Anweisung

```
i++;
```

addiert z. B. den Wert 1 zu der Variable i. Diese Operanden können Sie vor (Präfix-Notation) oder hinter den Operanden setzen (Postfix-Notation). Ein Unterschied wird allerdings erst dann sichtbar, wenn die Operatoren in Ausdrücken verwenden werden:

```
int i = 1;
Console.WriteLine(++i);
Console.WriteLine(i)
i = 1;
Console.WriteLine(i++);
Console.WriteLine(i);
```

Die Präfix-Notation bewirkt, dass der Compiler zuerst den Wert des Operanden addiert bzw. subtrahiert und danach den Ausdruck aus-

wertet. Der Ausdruck verwendet also den veränderten Wert. Die erste und die zweite `WriteLine`-Anweisung geben folglich den Wert 2 aus. Wenn Sie die Postfix-Notation verwenden, wird der Wert des Operanden erst addiert bzw. subtrahiert nachdem der Ausdruck vom Compiler ausgewertet wurde. Die dritte `WriteLine`-Anweisung gibt also 1 aus, die vierte gibt dann wieder 2 aus.

Ich halte diese, von C++ übernommene Auswertung der unären arithmetischen Operatoren für überflüssig, verwirrend und gefährlich. Sehr leicht können Sie in Ihren Programmen logische Fehler produzieren, wenn Sie diese Operatoren unbedacht einsetzen. Achten Sie immer darauf, dass die Postfix-Notation den Wert erst nach der Auswertung des Ausdrucks verändert, wenn Sie die Operatoren ++ und -- in Ausdrücken einsetzen. Idealerweise vermeiden Sie den Einsatz dieser Operatoren in Ausdrücken. Sie können die Addition/Subtraktion ja auch vor dem Ausdruck ausführen.

3.5.2 Bitoperationen

C# enthält einige Operatoren, über die Sie die einzelnen Bits eines Datentypen bearbeiten können.

Operator	Bedeutung
&	And
|	Or
^	XOr
~	Komplementär-Operation
<<	Linksverschiebung
>>	Rechtsverschiebung

Tabelle 3.13: Die bitweisen Operatoren

Im Ergebnis einer And-Operation sind nur die Bits gesetzt, die in dem einen *und* in dem anderen Operanden gesetzt sind. Im Dualsystem ausgedrückt ergibt z. B: 0101_2 `And` 0100_2 den Wert 0100_2, weil nur das dritte Bit (von rechts aus gesehen) in beiden Operanden gesetzt ist.

Den &-Operator benötigen Sie immer dann, wenn Sie überprüfen wollen, ob ein bestimmtes Bit in einem Datentypen gesetzt ist. Die folgende Anweisung überprüft z. B., ob das zweite Bit (das mit dem Wert 2) in der Variable i gesetzt ist:

```
int i = 3;
if ((i & 2) == 2)
   Console.WriteLine("Bit 2 ist gesetzt");
else
   Console.WriteLine("Bit 2 ist nicht gesetzt");
```

Mit Hilfe des Or-Operators können Sie einzelne Bits setzen. Im Ergebnis eine Or-Ausdrucks sind alle Bits gesetzt, die entweder in dem einen *oder* im anderen Operanden gesetzt sind. Dual ausgedrückt ergibt 0101_2 | 1001_2 z. B. den Wert 1101_2, weil die Bits 1 und 3 im ersten und die Bits 1 und 4 im zweiten Operanden gesetzt sind.

So können Sie z. B. gezielt das Bit 3 (das mit dem Wert 4) einer Variablen setzen, unabhängig vom bereits in der Variablen gespeicherten Wert:

```
i = i | 4;
```

Eine solche Zuweisung kann noch etwas kürzer geschrieben werden (vgl. Seite 134):

```
i |= 4;
```

Der XOr-Operator führt eine Exklusiv-Oder-Operation aus. Im Ergebnis ist ein Bit gesetzt, wenn dieses Bit in einem Operanden gesetzt und im anderen Operanden nicht gesetzt ist. Dual ausgedrückt ergibt z. B. 0101_2 ^ 1001_2 den Wert 1100_2. Mit diesem Operator können Sie einzelne Bits in einer Variablen gezielt umschalten. Gesetzte Bits werden so auf 0 gesetzt, Bits mit dem Wert 0 auf 1. Das folgende Beispiel schaltet das Bit 3 um:

```
i = i ^ 4;
```

Mit dem Komplementär-Operator können Sie das Einer-Komplement eines Wertes ermitteln. Dabei werden einfach alle Bits des Wertes umgekippt. Aus 1 wird 0 und aus 0 wird 1.

Mit den Operatoren >> und << können Sie die Bits eines Operanden um eine anzugebende Anzahl Stellen nach links oder nach rechts schieben. Die überstehenden Stellen werden dabei abgeschnitten, zur anderen Seite wird mit Nullen aufgefüllt. Wenn eine Byte-Variable den Wert 2 besitzt:

```
i = 2;
```

ist die duale Darstellung 00000010_2.

Linksschieben um eine Stelle:

```
i = i << 1;
```

ergibt die duale Zahl 00000100_2. Der Wert dieser Zahl ist jetzt 4. Linksschieben um eine Stelle entspricht einer Multiplikation mit 2. Rechtsschieben entspricht prinzipiell einer Ganzzahldivsion durch 2. Die rechten Bits fallen beim Schieben heraus, weswegen immer eine ganze Zahl resultiert.

Das Ergebnis einer Links-Schiebeoperation kann auch einen größeren Datentyp besitzen als der Operand. Ist der Operand z. B. vom Typ `byte` und speichert den Wert 128 (10000000_2) ergibt das Schieben nach Links keinen Überlauf, sondern den Wert 512 (100000000_2).

Fragen Sie mich bloß nicht, wofür man diese Bit-Schiebereien benötigt. Ganz früher, als die Rechner noch langsam waren, war eine Multiplikation mit einer Zweier-Potenz wesentlich schneller, wenn diese durch Linksschieben implementiert wurde: Die CPU stellt für solche Operationen direkte Befehle zur Verfügung. Heute sehe ich im kommerziellen Bereich eigentlich keine Anwendung mehr dafür. Möglicherweise benötigt die Mathematik solche Operationen. Aber davon habe ich keine Ahnung. Ich denke, die Schiebe-Operatoren sind hauptsächlich deswegen in C# enthalten, weil es diese auch in C++ gibt und schon immer gab.

3.5.3 Zuweisungen

Für Zuweisungen stellt C# nicht nur den Operator =, sondern auch noch einige spezielle Operatoren zur Verfügung. Eine einfache Zuweisung sieht z. B. so aus:

```
i = 1;
```

C# erlaubt auch eine Mehrfach-Zuweisung:

```
i = j  = 11;
```

i und *j* besitzen nach der Ausführung dieser Anweisung den Wert 11.

Die erweiterten Operatoren erlauben die Zuweisung eines Ausdrucks bzw. Werts, den Sie gleich noch über eine arithmetische Operation mit den Operanden berechnen. Die Anweisung

```
i += j;
```

addiert beispielsweise den Wert von j auf die Variable i und bedeutet soviel wie

```
i = i + j;
```

Operator	Bedeutung
=	einfache Zuweisung
+=	Additionszuweisung. `i += 1` entspricht `i = i + 1`.
-=	Subtraktionszuweisung. `i -= 1` entspricht `i = i - 1`.
*=	Multiplikationszuweisung. `i *= 1` entspricht `i = i * 1`.
/=	Divisionszuweisung. `i /= 1` entspricht `i = i / 1`.
%=	Modulo-Divisionszuweisung. `i %= 1` entspricht `i = i % 1`.
&=	Bitweise-And-Zuweisung. `i &= 1` entspricht `i = i & 1`.
|=	Bitweise-Or-Zuweisung. `i |= 1` entspricht `i = i | 1`.
^=	Bitweise-XOr-Zuweisung. `i ^= 1` entspricht `i = i ^ 1`.
<<=	Linksverschiebungszuweisung. `i <<= 1` entspricht `i = i << 1`.
>>=	Rechtsverschiebungszuweisung. `i >>= 1` entspricht `i = i >> 1`.

Tabelle 3.14: Die Zuweisungsoperatoren

3.5.4 Vergleiche

Für Vergleiche bietet C# die üblichen Operatoren. Ein Vergleichsausdruck ergibt immer den booleschen Wert `true` (wenn der Vergleich wahr ist) oder `false` (wenn der Vergleich falsch ist). Der Vergleich auf Gleichheit verwendet den Operator ==.

Operator	Beschreibung	gilt für Datentyp
==	Gleichheit	alle
!=	Ungleichheit	alle
<	Kleiner	nur für nummerische Typen und Aufzählungen (Enums)
>	Größer	nur für nummerische Typen und Aufzählungen
<=	Kleiner/Gleich	nur für nummerische Typen und Aufzählungen
>=	Größer/Gleich	nur für nummerische Typen und Aufzählungen

Tabelle 3.15: Die Vergleichsoperatoren

Nur die Operatoren == und != können Sie auf alle .NET-Datentypen anwenden. Dabei müssen Sie beachten, dass diese Operatoren nur bei Wertetypen, dem Typ `String` und bei Typen, die diese Operatoren speziell überladen haben (was das ist und wie das geht, zeigt Kapitel 4) einen Vergleich der gespeicherten Werte ergeben. Vergleichen Sie Referenztypen, vergleichen Sie prinzipiell daraufhin, ob diese dasselbe Objekt referenzieren. Referenzieren zwei Variablen unterschiedliche Objekte, die aber dieselben Werte speichern, ergibt ein Vergleich mit == prinzipiell `false`. Da die Vergleichsoperatoren für bestimmte Typen aber auch überladen werden können, kann es auch sein, dass ein Vergleich von Referenztypen doch die gespeicherten Werte miteinander vergleicht. Im .NET-Framework ist das zwar laut der Microsoft-Dokumentation nur für den Typ `String` der Fall, es kann aber immer (für neue Typen beispielsweise) vorkommen. Lesen Sie also die Dokumentation zu den verwendeten Typen oder probieren Sie den Vergleich einfach aus. Oder:

Vergleichen Sie Referenztypen immer über die von `object` *geerbte* `Equals`*-Methode. Dann kann eigentlich nichts passieren. Einige Typen wie* `String` *besitzen daneben eine* `CompareTo`*-Methode, über die Sie herausfinden können, ob der Typ kleiner, gleich oder größer ist als ein anderer. Die Rückgabe dieser Methode ist <o (der Typ ist kleiner), o (beide Typen sind gleich) oder >o (der Typ ist größer).*

Beim Vergleich von Strings berücksichtigt C# Groß- und Kleinschreibung. Der Vergleich "a" == "A" ergibt also `false`. Eine einfache Lösung dieses Problems ist, Strings einfach vor dem Vergleich in Klein- oder Großschrift umzuwandeln:

```
s1 = "ZAPHOD"; s2 = "zaphod";
   if (s1.ToLower() == s2.ToLower())
```

Alternativ können Sie auch die `Compare`-Methode der `String`-Klasse verwenden:

```
string s1 = "ZAPHOD"; string s2 = "zaphod";
if (String.Compare(s1, s2, true) == 0)
   Console.WriteLine("Beide Strings sind gleich");
```

Im dritten Argument geben Sie an, ob Sie Groß- und Kleinschreibung ignorieren wollen.

Die Operatoren >, <, >= und <= können Sie nur auf nummerische Datentypen und Aufzählungen (Enums) anwenden. Ärgerlich ist das schon, wenn Sie einmal zwei Strings miteinander vergleichen wollen. Warum Microsoft diese Operatoren nicht für den Datentyp `String` implementiert hat, ist absolut unverständlich. Sie können aber stattdessen die `CompareTo`-Methode verwenden. Viele Typen, die Vergleiche zulassen, besitzen diese Methode. Über die Rückgabe können Sie herausfinden, ob der Typ kleiner (Rückgabe < o), gleich (Rückgabe o) oder größer (Rückgabe > o) ist als ein anderer. So können Sie z. B zwei Strings daraufhin vergleichen, ob ein String größer ist als der andere:

```
s1 = "100"; s2 = "20";
if (s1.CompareTo(s2) > 0 )
   Console.WriteLine("s1 ist größer als s2");
else
   Console.WriteLine("s1 ist kleiner als s2");
```

Das Ergebnis wird Sie vielleicht erstaunen: "100" ist kleiner als "20". Das ist aber korrekt so, denn Strings werden immer Zeichen für Zeichen nach dem Unicode der Zeichen verglichen. Die linken Zeichen besitzen dabei eine höhere Priorität als die rechten. Da das erste Zeichen von *s2* einen höheren Wert besitzt, als das erste Zeichen von *s1*, ist *s2* folglich größer als *s1*. Darauf müssen Sie immer achten wenn Sie Strings vergleichen, die Zahlwerte speichern.

3.5.5 Logische Operatoren

Vergleichsausdrücke können Sie über die logischen Operatoren von C# miteinander verknüpfen. Der Ausdruck

```
Ausdruck1 && Ausdruck2
```

ergibt z. B. nur `true`, wenn beide Teilausdrücke `true` ergeben. Tabelle 3.16 zeigt die logischen Operatoren von C#.

Operator	Beschreibung
&&	Logisches And. Der rechte und der linke Ausdruck müssen `true` ergeben, damit der Gesamtausdruck `true` ergibt.
||	Logisches Or. Wenn einer der Ausdrücke `true` ergibt, resultiert `true` für den Gesamtausdruck.
!	Logisches Not. Der Ausdruck, der rechts von diesem Operator steht, wird negiert. Aus `true` wird `false` und aus `false` wird true.

Tabelle 3.16: Die logischen Operatoren

Wie in allen anderen Sprachen auch, besitzen diese Operatoren eine Priorität. ! wird vor && ausgewertet und && vor ||. Zur Sicherheit sollten Sie kombinierte logische Operationen immer klammern. Das gilt besonders dann, wenn Sie den !-Operator verwenden, der die höchste Priorität besitzt.

Der folgende Quellcode überprüft, ob jetzt gerade Samstag oder Sonntag und nach 12 Uhr ist:

```
DateTime date = DateTime.Now;
if (((date.DayOfWeek == DayOfWeek.Saturday) ||
```

```
   (date.DayOfWeek == DayOfWeek.Sunday)) &&
   (date.TimeOfDay.Hours > 12))
   Console.WriteLine("Jetzt ist Samstag oder " +
      "Sonntag nach 12 Uhr");
else
   Console.WriteLine("Jetzt ist nicht Samstag " +
      "oder Sonntag nach 12 Uhr");
```

C# wertet zusammengesetzte logische Ausdrücke von links nach rechts aus. Ergibt ein links stehender Ausdruck `false`, werden die rechts stehenden Ausdrücke nicht weiter ausgewertet. Das ist enorm hilfreich, wenn Teilausdrücke nur dann ausgewertet werden sollen, wenn andere Teilausdrücke `true` ergeben. Bei der Ermittlung eines Teilstrings aus einem String darf beispielsweise die Anzahl der extrahierten Zeichen die Länge des Strings nicht überschreiten:

```
string name = "";
if (name.Substring(0, 6) == "Zaphod") // erzeugt eine    // Ausnahme,
weil der String nicht genügend
// Zeichen speichert
```

Wenn Sie vor dem Extrahieren überprüfen, ob der String lang genug ist, wird keine Ausnahme erzeugt, weil der rechte Teilausdruck nicht mehr ausgewertet wird, wenn der String zu kurz ist:

```
string name = "";
if (name.Length >= 6 &&
   name.Substring(0, 6) == "Zaphod")
   Console.WriteLine("Hallo Zaphod");
```

3.5.6 ?:, sizeof, typeof und is

Der Bedingungsoperator ?:

Über den Bedingungsoperator ?: können Sie einen Ausdruck schreiben, der eine Bedingung überprüft und jeweils einen anderen Wert ergibt, wenn die Bedingung wahr oder falsch wird:

Bedingung ? *Ausdruck1* : *Ausdruck2*

Das folgende Beispiel überprüft, ob *x1* ungleich 0 ist und gibt in diesem Fall den Sinus von *x1* zurück, ansonsten 1:

```
double x1 = 0;
x1 != 0d ? Math.Sin(x1) : 1d;
```

Das Ergebnis eines solchen Ausdrucks können Sie in weiteren Ausdrücken verwenden oder einer Variablen zuweisen:

```
double x1 = 0; x2;
x2 = x1 != 0d ? Math.Sin(x1) : 1d;
```

Übersichtlicher wird dies, wenn Sie den Ausdruck klammern:

```
x2 = (x1 != 0d ? Math.Sin(x1) : 1d);
```

Vergleichen können Sie einen solchen Ausdruck mit einer if-Abfrage;

```
if (x1 != 0d)
   x2 = Math.Sin(x1);
else
   x2 = 1d;
```

Der Bedingungsoperator wird allerdings wesentlich schneller ausgeführt und ist einfacher anzuwenden.

Der sizeof-Operator

Über den `sizeof`-Operator können Sie die Größe eines Wertetypen ermitteln:

```
sizeof(Typ)
```

Dieser Operator kann nur in einem unsicheren Bereich des Quellcodes verwendet werden:

```
unsafe
{
    Console.WriteLine("int besitzt die Größe: " +
       sizeof(int));
   Console.WriteLine("DateTime besitzt die Größe: " +
      sizeof(DateTime));
}
```

Der Grund für die Verwendung in einem unsicheren Block ist, dass dieser Operator bereits zur Kompilierungszeit ausgewertet wird und dass die Größe von Datentypen sich in neueren Versionen des .NET-Framework ändern kann. Der `DateTime`-Typ kann z. B. in neuen Versio-

nen zwölf statt acht Byte groß sein. Ein altes Programm würde unter einer neuen .NET-Framework-Version dann fehlerhaft ausgeführt werden.

Verwenden Sie diesen Operator also idealerweise erst gar nicht. Wenn Sie nur mit den Klassen des .NET-Framework arbeiten, benötigen Sie `sizeof` nicht. Nur wenn Sie (Windows-)API-Funktionen direkt aufrufen (was nach der .NET-Philosophie niemals notwendig sein sollte), müssen Sie manchen dieser Funktionen Strukturen und deren Größe übergeben. Dann müssen Sie normalerweise `sizeof` einsetzen.

Der typeof-Operator

Über den `typeof`-Operator können Sie Informationen zu einem Typ ermitteln:

```
typeof(Typ)
```

`typeof` ergibt ein Objekt der Klasse `Type`. Dasselbe Objekt wird auch von der `GetType`-Methode zurückgegeben, die jede Instanz eines Typs besitzt. `typeof` wird aber nicht auf Instanzen, sondern auf die Typen direkt angewendet.

Die `Type`-Klasse besitzt eine große Anzahl an Eigenschaften und Methoden, über die Sie Informationen zum Typ ermitteln können. Die `Name`-Eigenschaft gibt z. B. den Namen des Typs zurück. Wenn Sie beispielsweise eine Klasse besitzen, die die Daten einer Person speichert:

```
class Person
{
   public string FirstName;
   public string SurName;
}
```

können Sie mit `typeof` Informationen zu dieser Klasse ermitteln:

```
Type t = typeof(Person);
Console.WriteLine("Name: " + t.Name);
Console.WriteLine("Assemblierung: "+ t.Assembly.FullName);
Console.WriteLine("Basistyp: "+ t.BaseType.Name);
Console.WriteLine("Voller Name: "+ t.FullName);
Console.WriteLine("Typ ist " +
```

```
   (t.IsClass ? "eine" : "keine" ) +
   " Klasse");
Console.WriteLine("Typ ist " +
   (t.IsArray ? "ein" : "kein") +
    " Array");

Console.WriteLine("Typ ist " +
   (t.IsEnum ? "eine" : "keine") +
   " Aufzählung");
Console.WriteLine("Typ ist " +
   (t.IsInterface ? "eine" : "keine") +
   " Schnittstelle");
```

Dieser Vorgang wird übrigens als Reflektion (Reflection) bezeichnet: Sie können unter .NET jederzeit Informationen zu allen Typen erhalten. Das funktioniert sogar dann, wenn diese Typen in externen Assemblierungen gespeichert sind, zu denen Sie keinen Quellcode besitzen. .NET stellt Ihnen dazu Klassen zur Verfügung, die Sie im Namensraum `System.Reflection` finden.

Der is-Operator

Über den `is`-Operator können Sie herausfinden, ob ein Ausdruck einen bestimmten Typ besitzt:

Ausdruck `is` *Typ*

Üblicherweise überprüfen Sie damit Variablen. Sinn macht das u. a., wenn Sie Daten in `object`-Typen verwalten (bei allen anderen Typen kennen Sie und der Compiler den gespeicherten Typ):

```
object o = (int)10;
if (o is int)
   Console.WriteLine("o ist ein int.");
else
   Console.WriteLine("o ist kein int.");
```

Sie können mit `is` jedoch auch überprüfen, ob ein Typ eine bestimmte Schnittstelle implementiert. Dazu vergleichen Sie einfach mit dem Typ der Schnittstelle. Schnittstellen werden im nächsten Kapitel behandelt.

3.6 Verzweigungen und Schleifen

C# kennt natürlich auch die gängigen Schleifen und Verzweigungen. Bevor ich die Verzweigungen und Schleifen beschreibe, möchte ich Ihnen einen wichtigen Tipp geben: Wenn Sie in Visual Studio programmieren und versehentlich eine Endlosschleife produziert haben (eine Schleife, die nie beendet wird), scheint Ihr Programm nicht mehr zu reagieren, weil der Prozessor in diesem Fall sehr stark ausgelastet ist. Statt das Programm über den Task-Manager »abzuschießen«, betätigen Sie lieber in Visual Studio einfach Strg + Pause um das Programm zu unterbrechen.

3.6.1 Die if-Verzweigung

Die `if`-Verzweigung besitzt die folgende Syntax:

```
if (Bedingung)
   Anweisungsblock1
[else
   Anweisungsblock2]
```

Wenn die Bedingung wahr wird, verzweigt das Programm in den Anweisungsblock 1. Wenn Sie den optionalen `else`-Block angeben, wird dieser ausgeführt, wenn die Bedingung falsch wird. Das folgende Beispiel überprüft, ob der Wert einer Variablen größer ist als der einer anderen:

```
if (x > y)
   Console.WriteLine("x ist größer als y.");
else
   Console.WriteLine("x ist kleiner oder gleich y.");
```

Wenn Sie mehrere Anweisungen in einem Block unterbringen wollen, müssen Sie diese in geschweifte Klammern einfügen, aber das wissen Sie ja bereits:

```
if (x > y)
{
   Console.WriteLine("x ist größer als y.");
   Console.WriteLine("y ist kleiner oder gleich x.");
}
else
{
```

```
   Console.WriteLine("x ist kleiner oder gleich y.");
   Console.WriteLine("y ist größer als x.");
}
```

Die Bedingung kann natürlich mit den logischen Operatoren auch komplex gestaltet werden. Das folgende Beispiel überprüft, ob *x* größer als *y* und *y* kleiner als *z* ist:

```
if (x > y && y < z)
   ...
```

3.6.2 Die switch-Verzweigung

Die `switch`-Verzweigung überprüft einen Ausdruck auf verschiedene anzugebende Werte und verzweigt in den `case`[1]-Block, der mit dem jeweiligen Wert assoziiert ist. Der optionale `default`-Block behandelt alle anderen Fälle.

```
switch (Prüfausdruck)
{
   case Wert1:
      /* Anweisungen für den Fall 1 */
      Sprunganweisung;

   [case Wert2:
      /* Anweisungen für den Fall 2 */
      Sprunganweisung;]

   [...]

   [default:
      /* Anweisungen, die ausgeführt werden sollen,
       * wenn keiner der Fälle eingetreten ist*/
      Sprunganweisung;]
}
```

Als Sprunganweisung geben Sie normalerweise `break` an. Diese Anweisung bewirkt, dass das Programm an der ersten Zeile hinter dem `switch`-Block weiter abgearbeitet wird. Alternativ ist die Verwendung der in Tabelle 3.17 dargestellten Sprunganweisungen möglich.

1. engl. für »Fall«

Sprunganweisung	Bedeutung
`break`	Sprung aus dem Block.
`continue`	Sprung zum nächsten Fall.
`default`	Sprung zum `default`-Block.
`goto` *`Label`*	Sprung zu einem Label, das am Anfang einer Zeile mit *`Label`*: definiert ist.
`return`	Sprung aus der Methode heraus.

Tabelle 3.17: Die C#-Sprunganweisungen

Das unselige, unstrukturierte `goto` sollten Sie besser nicht verwenden. Wenn Sie `goto` verwenden, wird der Softwareteufel Ihr Programm zerstören ...

Das folgende Beispiel lässt den Anwender eine Zahl eingeben und wertet diese dann aus:

```
Console.Write("Ihre Lieblingszahl: ");
int i = Convert.ToInt32(Console.ReadLine());

switch (i)
{
   case 7:
      Console.WriteLine("7 ist eine gute Zahl.");
      break;

   case 42:
      Console.WriteLine("Cool. Trinken Sie einen " +
         "pangalaktischen Donnergurgler.");
      break;

   case 3:
      Console.WriteLine("3 ist OK.");
      break;

   default:
      Console.WriteLine("Denken Sie noch einmal " +
         "darüber nach.");
      break;
}
```

Mehrere Anweisungen in den einzelnen Blöcken müssen Sie beim `switch` ausnahmsweise nicht in geschweifte Klammern einfügen. Das `break` in jedem Fall-Block bewirkt, dass die Verzweigung beendet wird und das Programm hinter der schließenden Klammer weiter ausgeführt wird.

Anders als einige andere Sprachen erlaubt C# nicht nur die Überprüfung einer einfachen nummerischen Variablen, sondern auch die von komplexen Ausdrücken, die sogar Zeichenketten ergeben können. Damit können Sie sogar Funktionen im Prüfausdruck aufrufen. Das folgende Beispiel ruft die `ReadLine`-Methode der `Console-Klasse` direkt im Prüfausdruck auf, um den Anwender seinen Namen eingeben zu lassen und diesen dann auszuwerten:

```
Console.Write("Ihr Name: ");
switch (Console.ReadLine())
{
   case "Zaphod":
      Console.WriteLine("Oh, hallo. " +
         "Wie geht es dem Weltall so?");
      break;
   case "b":
      case "Arthur":
      Console.WriteLine("Hat dein Haus " +
         "den Bagger überlebt?");
      break;
   default:
      Console.WriteLine("Guten Tag Fremder");
      break;
}
```


3.6.3 Die while-Schleife

Die `while`-Schleife überprüft die Schleifenbedingung im Kopf und wiederholt die enthaltenen Anweisungen so lange, wie diese Bedingung erfüllt ist:

```
while (Bedingung)
{
   Anweisungen
}
```

Beachten Sie, dass diese Schleife nicht mit einem Semikolon abgeschlossen werden muss.

Das folgende Beispiel zählt eine Variable hoch, solange diese einen Wert kleiner 10 besitzt:

```
int i = 1;
while(i < 10)
{
   Console.WriteLine(i);
   i++;
}
```

Eine `while`-Schleife können Sie mit `break` explizit abbrechen. So können Sie Schleifen erzeugen, die ihre Bedingung im Körper überprüfen:

```
i = 0;
while (true)
{
   i++;
   if (i > 9) break;
   Console.WriteLine(i);
}
```

Solche Schleifen benötigen Sie dann, wenn die Bedingung so komplex ist, dass diese nicht im Kopf oder Fuß der Schleife überprüft werden kann.

3.6.4 Die do-Schleife

Die `do`-Schleife überprüft die Bedingung im Fuß:

```
do
   Anweisungen
while (Bedingung);
```

Im Unterschied zur `while`-Schleife wird die `do`-Schleife mindestens einmal durchlaufen, auch wenn die Bedingung zu Anfang der Schleife bereits falsch ist. Das folgende Beispiel lässt den Anwender eine Antwort eingeben und schleift solange, bis dieser die »richtige« Antwort eingegeben hat:

```
string answer;
int i = 0;
do
{
   i++;
   if (i < 3)
      Console.Write("Geben Sie die Antwort ein: " );
   else
        Console.Write("Geben Sie die Antwort ein " +
           "(Tipp: Die richtige Antwort ist 42): " );
   answer = Console.ReadLine();
}
while (answer != "42");
```

Wie eine `while`-Schleife, können Sie eine `do`-Schleife explizit mit `break` abbrechen:

```
i = 0;
do
{
   i++;
   if (i > 9) break;
   Console.WriteLine(i);
} while (true);
```

3.6.5 Die for-Schleife

Die `for`-Schleife wird verwendet, wenn die Schleife eine festgelegte Anzahl Durchläufe besitzen soll. Im Kopf der Schleife initialisieren Sie die Zählvariable, geben an, bis zu welchem Wert diese gezählt werden soll und welchen Wert der Compiler in jedem Durchlauf der Schleife zu der Variable hinzuzählen oder von dieser abziehen soll:

```
for ([Initialisierungsausdruck]; [Bedingung];
   [Zählausdruck])
   Anweisungen
```

Der Initialisierungsausdruck kann eine Variablendeklaration enthalten.

Das folgende Beispiel zählt eine Variable von eins bis zehn in Einerschritten hoch:

```
for (int i=1; i<11; i++)
{
   Console.WriteLine(i);
}
```

Rückwärts-Zählen ist natürlich auch möglich:

```
for (int i=10; i>0; i--)
{
   Console.WriteLine(i);
}
```

Genauso können Sie auch um mehr als den Wert 1 hoch- oder herunterzählen.

3.7 Präprozessor-Direktiven

C# kennt einige Präprozessor-Direktiven, über die Sie die Kompilierung steuern können.

Direktive	Bedeutung
#define *Symbol* [= {true | false}]	Definition eines Bezeichners für die bedingte Kompilierung
#if *Bedingung* [#elif *Bedingung*] [#else] #endif	In einem solchen #if-Block können Sie Anweisungen unterbringen, die nur dann kompiliert werden, wenn die Bedingung wahr bzw. falsch wird. Die Bedingung kann alle Vergleichsoperatoren beinhalten und vergleicht mit den Symbolen, die mit #define angelegt wurden.
#warning *Warnung* #error *Fehler*	Diese Anweisung können Sie in einen #if-Block einbauen, um beim Eintritt einer Präprozessor-Bedingung eine Compiler-Warnung oder einen Compiler-Fehler zu generieren.
#region *Bezeichnung* #endregion	In diese Direktive schließen Sie Anweisungen ein, die als Region verwaltet werden sollen. In Visual Studio können Sie Regionen auf- und zuklappen.

Tabelle 3.18: Die wichtigsten C#-Präprozessor-Direktiven

Bedingte Kompilierung

Über einen #if-Block können Sie eine bedingte Kompilierung erreichen. Die Arbeitsweise entspricht der der `If`-Verzweigung, mit dem Unterschied, dass Code in Blöcken, deren Bedingung nicht zutrifft, vom Compiler nicht berücksichtigt wird. Dieser Code kann dann auch für die aktuelle Umgebung nicht benutzbare Anweisungen enthalten.

Der Bedingungsausdruck darf lediglich Literale, Operatoren und Symbole für bedingte Kompilierung enthalten. Diese speziellen Konstanten können Sie im Kopf der Datei Modul über `#define` *Name*`[= {true | false}]` einrichten.

Interessant ist die Verwendung der bedingten Kompilierung für die Kompilierung einer einer eingeschränkten Shareware- oder Testversion einer Anwendung:

```
#define DEBUG // implizit true
#define SHAREWARE // implizit true

using System;

namespace Nitty_Gritty.Samples.Präprozessor_Direktiven
{
   class Start
   {
      static void Print()
      {
        #if SHAREWARE
         Console.WriteLine("Die Shareware-Version " +
            "erlaubt keinen Ausdruck.");
         return;
        #else
          Console.WriteLine("Das ist der Druck ...");
        #endif
      }

      [STAThread]
      static void Main(string[] args)
      {
         Console.WriteLine("Präprozessor-Direktiven");
         #if (DEBUG && SHAREWARE)
           Console.WriteLine("Einige Debug-Ausgaben " +
```

```
            " für die Shareware-Version ...");
        #elif (DEBUG)
          Console.WriteLine("Einige Debug-Ausgaben " +
            " für die normale Version ...");
        #endif

          Print();
      }
   }
}
```

3.8 Reflektion und Attribute

Reflektion

Reflektion erlaubt, dass ein Programm Informationen zu den Typen einer Assemblierung auslesen kann. Das .NET-Framework stellt dazu Klassen zur Verfügung, die im Namensraum `System.Reflection` verwaltet werden. Ein Teilbereich der Reflektion erlaubt sogar, Assemblierungen und deren Typen in der Laufzeit dynamisch zu erzeugen. Entwicklungsumgebungen wie Visual Studio und SharpDevelop nutzen Reflektion, um Informationen zu den Typen zu ermitteln und auszugeben, die Sie im Programm verwenden. Das IntelliSense-Feature von Visual Studio basiert z. B auf Reflektion.

Reflektion ist ein mächtiger Mechanismus, den ich hier nur grundlegend beschreiben kann. Der `Reflection`-Namensraum enthält ca. 40 Klassen und Schnittstellen, mit denen Sie arbeiten können.

Einen Teil der Reflektion nutzen Sie bereits, wenn Sie mit dem `typeof`-`Operator` oder über die `GetType`-Methode eines Typs Informationen zum Typ auslesen. Sie können aber auch die Typen anderer Assemblierungen auslesen. Der folgende Quellcode zeigt, wie Sie dies prinzipiell machen:

```
System.Reflection.Assembly assembly;

/* Ermitteln der Assemblierung, in der der Typ
 * string gespeichert ist */
assembly = System.Reflection.Assembly.GetAssembly(
   typeof(string));
```

```
/* Alternativ können Sie die Assemblierung auch
   * aus einer Datei laden */
//string filename = @"C:\WINNT\Microsoft.NET\" +
//   "Framework\v1.0.3215\System.dll";
//assembly = System.Reflection.Assembly.LoadFrom(
//   filename);

/* Alle Typen auslesen */
foreach (Type t in assembly.GetTypes())
{
   /* Informationen zum Typ ausgeben */
   Console.Write(t.Name + ": ");
   Console.Write(t.IsPublic ? "Public " : "Private ");
   if (t.IsArray) Console.WriteLine("Array");
   if (t.IsClass)
   {
      Console.WriteLine("Class");
      /* Methoden auslesen */
      Console.WriteLine("   Methoden");
      foreach (System.Reflection.MethodInfo
         m in t.GetMethods())
      {
         Console.WriteLine("   " + m.Name);
       }
   }
   if (t.IsEnum) Console.WriteLine("Enum");
   if (t.IsInterface) Console.WriteLine("Interface");
   Console.WriteLine();
}
```

Attribute

Über Attribute können Sie Typen oder Elemente dieser Typen (Eigenschaften, Methoden) beschreiben. Über Reflektion können diese Attribute ausgelesen werden. Einige Attribute werden aber auch vom Compiler verwendet. Das Attribut `WebMethod` definiert z. B. eine Methode als Web-Methode (die in einem Webdienst verwendet wird).

Attribute sind in Attributklassen definiert. Das .NET-Framework definiert einige Attribute vor. Einige der wichtigsten stellt Tabelle 3.19 dar. Wenn Sie eines dieser Attribute verwenden wollen, setzen Sie diese in eckigen Klammern vor die Deklaration und übergeben dem Attribut in Klammern eventuelle Argumente.

Attribut	Beschreibung
`Conditional(` `string` *`conditionString`*`)`	markiert ein Element als nur zu kompilieren, wenn das als String angegebene (mit `#define` definierte) Präprozessor-Symbol den Wert `true` besitzt. Wenn Sie dieses Element im Quellcode verwenden und das Präprozessor-Symbol ist `false` oder nicht vorhanden, erzeugt die CLR keine Ausnahme, sondern führt die entsprechende Anweisung einfach nicht aus. Damit können Sie z. B. ganz einfach Debug-Ausgaben im Programm ermöglichen, deren Anweisungen Sie nicht unbedingt im Release der Anwendung entfernen müssen. Beachten Sie, dass dieses Attribut im `System.Diagnostics`-Namensraum verwaltet wird.
`Obsolete(` `string` *`message`* `[, bool` *`isError`*`])`	markiert einen Typ oder ein Element als überholt. Bei der Verwendung generiert der Compiler eine Warnung, wenn Sie *`isError`* nicht angeben. Geben Sie `true` in *`isError`* an, erzeugt der Compiler einen Fehler.

Tabelle 3.19: Die wichtigsten Attribute

Das folgende Beispiel verwendet das Attribut `Conditional` für Debugausgaben und `Obsolete`, um eine Klasse als veraltet zu kennzeichnen:

```
#define DEBUG

using System;

namespace Nitty_Gritty.Samples.Attribute
{

   [Obsolete("Verwenden Sie die Klasse XPrinter")]
   public class Printer
   {
      /* ... */
   }
```

```
   public class XPrinter
   {
      /* ... */
   }


   public class Start
   {

      [System.Diagnostics.Conditional("DEBUG")]
      static void TraceOutput(string message)
      {
         Console.WriteLine(message);
      }

      [STAThread]
      public static void Main()
      {
         /* Hier generiert der Compiler eine Warnung */
         Printer p = new Printer();
         /* ... */

         /* Methode verwenden, die nur kompiliert wird,
          * wenn das Präprozessor-Symbol
          * DEBUG true ist */
         TraceOutput("Irgendeine Debug-Info ...");
      }
   }
}
```

Das `Obsolete`-Attribut ist in globalen Klassenbibliotheken sinnvoll, die bereits von Anwendungen werden und die Sie nachträglich um neue Klassen, Methoden oder Eigenschaften erweitern. Da alte Anwendungen die alten Elemente verwenden, müssen Sie diese in der Klassenbibliothek belassen. Kennzeichnen Sie diese dann für die Entwicklung neuer Anwendungen als veraltet.

Sie können auch eigene Attributklassen deklarieren um selbst definierte Attribute verwenden zu können. Diese müssen Sie von der Klasse `System.Attribute` ableiten. Die Beschreibung dieser Technik würde den Rahmen dieses Buchs sprengen.

4 Objektorientierte Programmierung

Dieses Kapitel beschreibt, wie Sie die OOP-Features von C# einsetzen. Die Grundlagen werden aber (natürlich) nicht beschrieben. Lesen Sie dazu den Artikel »OOP-Grundlagen« auf der Nitty-Gritty-Website. Dort werden die Grundlagen sehr ausführlich behandelt.

Die Beispielprogramme in diesem Kapitel habe ich ausnahmsweise größtenteils mit deutschsprachigen Bezeichnern entwickelt, obwohl ich eigentlich lieber englischsprachige Begriffe verwende (um zu den Sprachelementen konsistent zu sein). Ich wollte damit sicherstellen, dass das komplexe Thema OOP nicht noch durch teilweise schwierig zu übersetzende Begriffe unnötig erschwert wird.

4.1 Klassen und Strukturen

C# erlaubt die Deklaration von Klassen und Strukturen zur Strukturierung eines Programms. Klassen und Strukturen unterscheiden sich in C# prinzipiell kaum. In beiden können Sie Eigenschaften und Methoden unterbringen. Für einige Leser werden Methoden in Strukturen (die in klassischen Programmen nur Daten speichern) etwas ungewöhnlich sein. C# definiert den Begriff Struktur aber etwas anders als gewohnt. Die folgende Auflistung macht die Unterschiede zwischen Klassen und Strukturen deutlich:

- Klassen werden mit dem Schlüsselwort `class` deklariert, Strukturen mit `struct`.
- Instanzen von Strukturen sind Wertetypen, Instanzen von Klassen sind Referenztypen. Die Arbeit mit Strukturinstanzen kann wesentlich performanter sein als die mit Klassen (weil keine Referenzen aufgelöst werden müssen). Wenn Sie Strukturen allerdings an Methoden oder Objekte übergeben, von Methoden zurückgeben oder ähnliches, werden die Werte kopiert (was bei Referenztypen nicht

der Fall ist). Das kann dann wieder zu einem Performanceverlust führen. Verwenden Sie Strukturen also immer dann, wenn Sie diese nur in einer Methode benötigen und nicht weitergeben müssen.

- Klassen unterstützen Vererbung und Schnittstellen, Strukturen können lediglich Schnittstellen implementieren.
- Klassen können Destruktoren besitzen, Strukturen nicht.
- In Klassen können Sie einen parameterlosen Default-Konstruktor unterbringen. Strukturen erlauben dies nicht. Der vordefinierte Default-Konstruktor einer Struktur initialisiert alle Werte der Eigenschaften auf einen Leerwert (o bei nummerischen Typen, "" bei einem String etc.), was bei Klassen nicht der Fall ist. Strukturen erlauben wie Klassen Konstruktoren mit Argumenten. Sie müssen allerdings im Gegensatz zu Klassen in solchen Konstruktoren alle Datenfelder initialisieren.
- Strukturen erlauben nicht, dass Datenfelder bei der Deklaration initialisiert werden.

Strukturen werden in diesem Buch nicht weiter behandelt. Wenn Sie die Unterschiede im Kopf behalten, können Sie Strukturen wie Klassen deklarieren. Das folgende Beispiel deklariert eine Struktur zur Speicherung von Personendaten:


```
public struct Person
{
   /* Einige einfache Eigenschaften */
   public string Vorname;
   public string Nachname;
   public string Strasse;
   public int Postleitzahl;
   public string Ort;

   /* Eine einfache Methode, die den vollen Namen
    * der Person zurückgibt */
   public string VollerName()
   {
      return Vorname + " " + Nachname;
   }
}
```

Strukturen müssen, obwohl es sich um Wertetypen handelt, mit `new` *instanziert werden.*

Instanzieren und Verwenden einer Struktur:

```
Person p = new Person();
p.Vorname = "Fred Bogus";
p.Nachname = "Trumper";
Console.WriteLine(p.VollerName());
```

4.2 Die Strukturierung einer Anwendung

Größere Programme erfordern eine Strukturierung des Quellcodes. Die Basis der Strukturierung ist in C# eine Klasse oder eine Struktur. Über Klassen und Strukturen erreichen Sie, dass Sie Programmcode wiederverwenden können und dass Sie Ihr Programm so strukturieren, dass die Fehlersuche und die Wartung erheblich erleichtert wird.

Um nicht immer doppelgleisig fahren zu müssen, verwende ich ab hier nur noch den Begriff »Klasse«. Wenn Sie aber die Unterschiede beachten, die ich auf Seite 155 beschrieben habe, können Sie alles genannte auch auf Strukturen anwenden.

Wenn Sie eine Klasse entwickeln, können Sie entscheiden, ob Sie eine »echte« Klasse (mit »normalen« Eigenschaften und Methoden) programmieren, aus der Sie später Instanzen erzeugen, oder ob Sie eine Klasse mit statischen Methoden und Eigenschaften erzeugen wollen. Statische Methoden und Eigenschaften (Klassenmethoden, Klasseneigenschaften) können Sie auch ohne eine Instanz der Klasse aufrufen. »Echte« Klassen arbeiten echt objektorientiert, Klassen mit statischen Methoden und Eigenschaften simulieren (u. a.) die Module der strukturierten Programmierung.

Die besten Beispiele für statische Methoden finden Sie in den Klassen des .NET-Framework. Viele der dort enthaltenen Methoden können Sie verwenden, ohne eine Instanz der Klasse zu erzeugen.

Statische Methoden und Eigenschaften werden ab Seite 182 beschrieben. Zunächst behandelt dieses Kapitel ganz normale Klassen, aus denen Sie Instanzen erzeugen müssen.

4.3 Einfache Klassen und deren Anwendung

Einfache Klassen deklarieren

In einer C#-Datei (mit der Endung .CS) können Sie eine oder mehrere Klassen implementieren. In Visual Studio fügen Sie Ihrem Projekt dazu eine neue *cs*-Datei über den Befehl KLASSE HINZUFÜGEN im Menü PROJEKT hinzu. Visual Studio erzeugt in der Datei eine neue leere Klasse, die denselben Namen trägt wie die Datei. Sie können den Klassennamen natürlich auch ändern.

Eine einfache Klasse zur Speicherung von Personendaten wird z. B. folgendermaßen deklariert:

```
public class Person
{
   /* Einige einfache Eigenschaften */
   public string Vorname;
   public string Nachname;
   public string Strasse;
   public int Postleitzahl;
   public string Ort;

   /* Eine einfache Methode, die den vollen Namen
    * der Person zurückgibt */
   public string VollerName()
   {
      return Vorname + " " + Nachname;
   }
}
```

Die Klasse und deren Elemente sind mit dem Modifizierer `public` deklariert, damit diese von außen verwendet werden können.

Objekte instanzieren

Wenn Sie Klassen verwenden wollen, die keine statischen Elemente enthalten (die ab Seite 182 beschrieben werden), müssen Sie dazu zunächst eine Instanz dieser Klasse erzeugen. Dazu deklarieren Sie eine Variable vom Typ dieser Klasse und erzeugen das Objekt mit dem `new`-Operator:

```
public class Start
{
   [STAThread]
   public static void Main(string[] args)
   {
      /* Ein Objekt der Klasse Person erzeugen */
      Person p =  new Person();

      /* Das Objekt verwenden */
      p.Vorname = "Fred Bogus";
      p.Nachname = "Trumper";

      /* Eine Methode des Objekts verwenden */
      Console.WriteLine(p.VollerName());
   }
}
```

Aus einer Klasse können Sie so viele Objekte erzeugen, wie Sie benötigen. Jedes Objekt wird separat von den anderen Objekten gespeichert und besitzt seine eigenen Eigenschaftswerte.

Beachten Sie, dass die Instanzen von Klassen immer Referenztypen sind. Wenn Sie zwei Variablen aufeinander zuweisen, die Klasseninstanzen verwalten, kopieren Sie die Referenzen, nicht die Werte der Objekte. Wenn Sie eine Klasseninstanz an eine Methode übergeben, arbeitet die Methode über die übergebene Referenz mit der Instanz der Klasse.

4.4 Die Sichtbarkeit von Klassen und Klassenelementen

Klassen und deren Elemente können Sie in der Sichtbarkeit über Modifizierer einschränken. Tabelle 4.1 beschreibt die Bedeutung dieser Modifizierer für Klassen.

Modifizierer	Bedeutung
`internal`	legt fest, dass eine Klasse nur innerhalb der Assemblierung verwendet werden kann. Diesen Modifizierer verwenden Sie hauptsächlich in Klassenbibliotheken für Hilfsklassen, die Sie nicht veröffentlichen wollen.
`public`	legt fest, dass eine Klasse innerhalb der Assemblierung und von anderen Assemblierungen aus, die diese Assemblierung referezieren, verwendet werden kann.

Tabelle 4.1: Die Modifizierer für die Sichtbarkeit von Klassen

Den Modifizierer `private` können Sie für Klassen übrigens nur verwenden, wenn diese Bestandteil einer anderen Klasse sind (dann handelt es sich bei der inneren Klasse um ein Klassenelement der äußeren Klasse). Verschachtelte Klassen werden auf Seite 187 behandelt.

Tabelle 4.2 beschreibt die Bedeutung der Sichtbarkeitsmodifizierer für Klassenelemente

Modifizierer	Bedeutung
`private`	Mit `private` stellen Sie die Sichtbarkeit eines Elements so ein, dass dieses nur innerhalb der Klasse gilt. Solche Elemente können von außen nicht verwendet werden. Wenn Sie damit eine Eigenschaft deklarieren, handelt es sich im Prinzip dabei um einfache Variablen, die von allen Methoden der Klasse verwendet, aber nicht von außen gesetzt oder gelesen werden können.
`protected`	Der Modifizierer `protected` kennzeichnet Klassenelemente, die zunächst wie private Eigenschaften auftreten, aber bei der Vererbung (ab Seite) in abgeleiteten Klassen verwendet werden können (was bei privaten Elementen eben nicht möglich ist).

Modifizierer	Bedeutung
`internal`	`internal`-Elemente gelten innerhalb der Assemblierung wie `public`-Elemente, verhalten sich nach außen aber wie `private`-Elemente. Mit diesem hauptsächlich in Klassenbibliotheken verwendeten Sichtbarkeitsbereich legen Sie fest, dass bestimmte Elemente nur innerhalb der Klassenbibliothek verwendet werden dürfen.
`protected internal`	Elemente, die mit `protected internal` gekennzeichnet sind, verhalten sich innerhalb der Assemblierung wie `protected`-Elemente und nach außen wir `private`-Elemente. Klassen, die innerhalb der Assemblierung von dieser Klasse abgeleitet werden, können also auf diese Elemente zugreifen.
`public`	Elemente, die den Modifizierer `public` besitzen, können von außen (über eine Referenz auf ein Objekt dieser Klasse oder – bei statischen Elementen – über den Klassennamen) verwendet werden.

Tabelle 4.2: Die Modifizierer für die Sichtbarkeit von Klassenelementen

4.5 Eigenschaften

4.5.1 Einfache Eigenschaften

Einfache Eigenschaften, die oft auch als »Felder« bezeichnet werden, deklarieren Sie wie Variablen, die ich bereits in Kapitel 3 beschrieben habe. Zur Einstellung der Sichtbarkeit verwenden Sie die auf Seite 160 beschriebenen Modifizierer.

Öffentliche Eigenschaften können von außen gesetzt und gelesen werden. Private Eigenschaften haben den Sinn, Werte über die Lebensdauer des Objekts zu speichern, und diese Werte allen Methoden (Funktionen) des Objekts zugänglich zu machen. Von außen können Sie private Eigenschaften nicht verwenden.

Eine Klasse zur Verwaltung von Kreisdaten verwaltet beispielsweise einen Wert für PI in einer privaten Eigenschaft und erlaubt das Setzen des Radius über eine öffentliche Eigenschaft:

```
public class Kreis
{
    /* Eine private Eigenschaft speichert den Wert
     * von PI (nur zu Demozwecken; den Wert von
     * PI können Sie aus Math.PI auslesen) */
    private double PI = 3.1415927;

    /* Eine öffentliche Eigenschaft speichert
     * den Radius */
    public double Radius;

    /* Eine Methode berechnet den Umfang */
    public double Umfang()
    {
        return Radius * 2 * PI;
    }
}
```

4.5.2 Kapselung

Die Kapselung ist ein wichtiges Grundkonzept der OOP. Kapselung bedeutet, dass Objekte den Zugriff auf ihre Daten kontrollieren.

Besitzt ein Objekt nur einfache Eigenschaften, so kann es den Zugriff auf seine Daten nicht kontrollieren. Ein Programmierer, der dieses Objekt benutzt, kann in die Eigenschaften hineinschreiben, was immer auch zu dem Datentyp der Eigenschaften passt. Das kann dazu führen, dass das Objekt ungültige Daten speichert und damit u. U. beim Aufruf von Methoden fehlerhaft reagiert. Die Kapselung verhindert einen solchen ungültigen Status des Objekts.

In C# können Sie Kapselung recht einfach erreichen, indem Sie die Eigenschaft mit Zugriffsmethoden[1] erweitern.

Eigenschaften mit Zugriffsmethoden

Eigenschaften können so erweitert werden, dass beim Schreiben und Lesen automatisch eine interne Zugriffsmethode aufgerufen wird. In dieser können Sie beliebig programmieren. Sie können zum Beispiel

1. Im Original werden diese als »Accessors« bezeichnet.

beim Schreiben überprüfen, ob der geschriebene Wert gültig ist, und beim Lesen den zurückzugebenden Wert vor der Rückgabe berechnen. Anders als in manchen anderen objektorientierten Sprachen können Sie in C# die Zugriffsmethoden so in die Eigenschaft integrieren, dass diese von außen wie eine ganz normale Eigenschaft wirkt (in der klassischen OOP würden Sie für solche Eigenschaften zwei separate echte Methoden – eine zum Schreiben und eine zum Lesen des Werts – implementieren).

Die für Eigenschaften verwendeten Begriffe sind in der OOP recht unterschiedlich. Viele Publikationen benennen einfache Eigenschaften als »Felder« (»Fields«). Eigenschaften mit Zugriffsmethoden werden im englischen Sprachraum oft als »Property« bezeichnet, was aber eigentlich nur der englische Begriff von »Eigenschaft« ist. Ich habe mich für den Begriff »Eigenschaft mit Zugriffsmethoden« entschieden, da der Begriff »Eigenschaft« in der OOP im Allgemeinen für die Datenfelder eines Objekts steht.

Für eine Eigenschaft mit Zugriffsmethoden benötigen Sie eine private Variable, die den Wert speichert. Dann deklarieren Sie die `set`- und die `get`-Zugriffsmethode. In der `set`-Zugriffsmethode wird der geschriebene Wert im impliziten Argument `value` übergeben. `get` ist eine Funktion, die den Wert zurückgibt. Das folgende Beispiel deklariert eine Klasse für Kreisberechnungen, die den Zugriff auf die Eigenschaft *Radius* so kapselt, dass diese keine negativen Werte zulässt.

```
public class Kreis
{
   /* Die Eigenschaft Radius kapselt ihren Wert */
   private double radius;
   public double Radius
   {
      /* Die set-Zugriffsmethode ist für das Schreiben
       * zuständig, hier wird der Wert auf Gültigkeit
       * überprüft */
      set
      {
```

```
            if (value >= 0)
               radius = value;
            else
               /* Wenn der Wert ungültig ist wird eine
                * Ausnahme erzeugt */
               throw new Exception("Der Radius kann " +
                  "nicht negativ sein");
         }

         /* Die get-Zugriffsmethode gibt den Wert der
          * Eigenschaft Radius zurück */
         get
         {
            return radius;
         }
      }

      /* Eine Methode berechnet den Umfang */
      public double Umfang()
      {
         return Radius * 2 * Math.PI;
      }
}
```

Dieses Beispiel erzeugt eine Ausnahme wenn ein ungültiger Wert geschrieben wird. Wenn Sie Klassen erzeugen, die ihre Eigenschaften kapseln, müssen Sie eigentlich immer eine Ausnahme erzeugen, wenn ein falscher Wert in die Eigenschaft geschrieben wurde. Nur über eine Ausnahme erfährt der Programmierer, der Ihre Klassen verwendet, auch wirklich, dass die Eigenschaft den Wert abgewiesen hat. Ein Programm sollte Ausnahmen grundsätzlich abfangen. So erhält dann auch der Anwender eine Meldung darüber, das etwas passiert ist. Das Erzeugen und Abfangen von Ausnahmen wird in Kapitel 6 behandelt. Um das Beispiel zu vervollständigen, folgt aber noch der Quellcode zur Erzeugung und Verwendung des Objekts, der die mögliche Ausnahme in einem `try-catch`-Block abfängt:

```
try
{
```

```
   /* Ein Objekt der Klasse Kreis erzeugen */
   Kreis k;
   k = new Kreis();

   /* Versuch, einem ungültigen Wert in die
    * Eigenschaft Radius zu schreiben */
   k.Radius = -100;

   /* und verwenden */
   Console.WriteLine("Ein Kreis mit {0} mm " +
      "Durchmesser besitzt {1} mm^2 Fläche", k.Radius,
      k.Umfang().ToString());
}

catch(Exception e)
{
   /* Hier werden alle Ausnahmen behandelt */
   Console.WriteLine(e.Message);
}
```

Schreibt ein Programm nun einen ungültigen Wert in dieses Objekt, erzeugt das Objekt die Ausnahme und das Programm fängt diese ab.

Eigenschaften mit Zugriffsmethoden werden allerdings nicht nur für Kapselung verwendet, sondern auch in spezifischen Programmierersituationen. Objekte, die eine grafische Darstellung besitzen, müssen z. B. beim Schreiben einer Eigenschaft, die das Aussehen beschreibt, eine Neuzeichnung initiieren. Diese Objekte rufen die Methode zum Zeichnen der Oberfläche dann in der `set`-Zugriffsmethode der Eigenschaft auf.

4.5.3 Schreibgeschützte, lesegeschützte und konstante Eigenschaften

Schreibgeschützte Eigenschaften

In vielen Situationen macht es Sinn, dass Eigenschaften nur gelesen werden können. Die Eigenschaft *Alter* einer Person kann zum Beispiel (leider) nicht geändert werden, sondern berechnet sich aus dem Geburtsdatum.

Einfache schreibgeschützte Eigenschaften erhalten Sie mit dem Modifizierer `readonly`. Solche Eigenschaften können Sie aber nur im Konstruktor der Klasse beschreiben oder bei der Deklaration initialisieren. Für das Alter einer Person eignet sich eine einfache schreibgeschützte Eigenschaft nicht, weil das Alter ja berechnet werden muss. Eine Klasse, die ein Konto repräsentiert, könnte die Kontonummer aber in einer solchen Eigenschaft verwalten, deren Wert im Konstruktor übergeben wird:

```
public class Konto
{
   public readonly int Nummer;
   public Konto(int nummer)
   {
      this.Nummer = nummer;
   }
}
```

Konstruktoren werden ab Seite 176 behandelt.

Wesentlich mehr Flexibilität besitzen Sie, wenn Sie Eigenschaften mit Zugriffsmethoden verwenden. Wenn Sie die `set`-Zugriffsmethode einfach weglassen, erhalten Sie eine schreibgeschützte Eigenschaft. Die *Person*-Klasse kann so mit einer Eigenschaft *Alter* erweitert werden, die nur gelesen werden kann und ihren Wert dabei berechnet:

```
   /* Eine schreibgeschützte Eigenschaft
    * berechnet das Alter der Person */
   public int Alter
   {
      get
      {
         int alter;
         alter = DateTime.Now.Year - Geburtsdatum.Year;
         if ((DateTime.Now.Month < Geburtsdatum.Month)
          | ((DateTime.Now.Month == Geburtsdatum.Month)
          & (DateTime.Now.Day < Geburtsdatum.Day)))
         {
            alter -= 1;
         }
```

```
        return alter;
      }
   }
```

Mit einem Objekt der Klasse *Person* können Sie nun das Alter auslesen, aber nicht beschreiben:

```
/* Ein Objekt der Klasse Person erzeugen */
Person p = new Person();

/* Das Objekt initialisieren */
p.Vorname = "Leonardo";
p.Nachname = "Da Vinci";
p.Geburtsdatum = new DateTime(1452,4,15);

/* Den Versuch, die schreibgeschützte
 * Eigenschaft Alter zu beschreiben,
 * verhindert der Compiler */
p.Alter = 100; // Fehler

/* Die Eigenschaft Alter auslesen */
Console.WriteLine(p.Vorname + " ist " +
   p.Alter.ToString() + " Jahre alt");
```

Testen Sie dieses Beispiel nicht mit Ihrem eigenen Geburtsdatum. Ich habe das gemacht und musste erkennen, wie alt ich eigentlich schon bin, worauf ich in eine etwa fünf Sekunden dauernde Depression verfiel.

Lesegeschützte Eigenschaften

Lassen Sie bei einer Eigenschaft die `get`-Zugriffsmethode weg, ist diese Eigenschaft lesegeschützt. Obwohl solche Eigenschaften eigentlich selten Sinn machen (ein Programm, das einen Wert in eine Eigenschaft schreiben kann, sollte diesen wohl auch lesen können), sind sie trotzdem möglich.

Konstante Eigenschaften

Eigenschaften können auch als Konstante deklariert werden. Dann können Sie den Wert der Eigenschaft nur in der Deklaration zuweisen und später nicht mehr ändern:

```
public const int MaximalesAlter = 150;
```

4.5.4 Indizierer

Wenn Sie eine Klasse entwickeln, die eine Auflistung implementiert (siehe Kapitel 5), können Sie über Indizierer einen intuitiven Zugriff auf die gespeicherten Elemente ermöglichen. Implementiert die Klasse einen Indizierer, können Sie wie bei einem Array auf die gespeicherten Elemente zugreifen. Indizierer erlauben aber auch andere und mehrere Datentypen als Schlüssel. Wenn Ihre Klasse beispielsweise eine Dictionary-Auflistung implementiert (Kapitel 5), können Sie den Zugriff über einen Integer-Index und einen String-Schlüssel ermöglichen.

Indizierer werden nach dem folgenden Schema deklariert:

```
[Attribute] [Modifizierer] Typ this [Parameterliste]
   {Zugriffsmethodenliste}
```

In der Parameterliste geben Sie die verschiedenen möglichen Zugriffs-Typen an. Ein Parameter besitzt die folgende Syntax:

```
[Attribute] Typ Bezeichner
```

In der *Zugriffsmethodenliste* deklarieren Sie für jeden Parameter eine `set`-/`get`-Zugriffsmethode. Sie können `set` oder `get` auch weglassen, um schreib- oder lesegeschützte Indizierer zu erzeugen.

Das folgende Beispiel zeigt eine einfache Klasse, die lediglich drei `double`-Werte speichert und deren Mittelwert berechnet:

```
public class Wertung
{
   private double[] einzelwertungen = new double[3];

   /* Indizierer */
   public double this[int index]
   {
```

```
      get {return einzelwertungen[index];}
      set {einzelwertungen[index] = value;}
   }

   /* Methode zur Berechnung des Mittelwerts */
   public double Mittelwert()
   {
      double result = 0;
      for (int i=0; i<3; i++)
         result += einzelwertungen[i];
      return result / 3;
   }
}
```

Nun können Sie über den Indizierer auf die gespeicherten Zahlen zugreifen:

```
/* Einzelwertungen speichern */
Wertung wertung = new Wertung();

/* Über den Indizierer auf die Elemente zugreifen */
wertung[0] = 8.3;
wertung[1] = 9.9;
wertung[2] = 9.2;

/* Mittelwert berechnen */
Console.WriteLine("Mittelwert: " +
   wertung.Mittelwert());
```

In den Buchbeispielen habe ich eine weitere Klasse implementiert, die eine Lottoziehung darstellt und den lesenden Zugriff auf die zufällig ermittelten Lottozahlen demonstriert.

4.6 Methoden

Methoden sind die Basis der Wiederverwendung von Programmcode. Immer dann, wenn Sie merken, dass Sie Programmcode in identischer oder ähnlicher Form an mehreren Stellen eines Pro-

gramms verwenden, sollten Sie diesen in eine Methode packen und die Methode (über eine Instanz der Klasse, oder, wenn es eine statische Methode ist, direkt) aufrufen.

C# kennt nur Methoden, die Werte zurückgeben. Eine Methode, die prinzipiell keinen Wert zurückgeben soll, wird mit dem Datentyp `void` (Leer) deklariert.

4.6.1 Deklaration

Methoden deklarieren Sie nach dem folgenden Schema:

```
[Attribute] [Modifizierer] Typ Name([Argumentliste])
{
   Anweisungen
   [return Ausdruck;]
}
```

Methoden geben immer einen Typ zurück, den Sie nach dem optionalen Modifizierer angeben. Methoden können jeden verfügbaren Typ zurückgeben, auch Objekte und Arrays. Wenn Sie eine Methode schreiben, die eigentlich nichts zurückgeben soll, deklarieren Sie diese mit dem Typ `void`.

In der Argumentliste können Sie ein oder mehrere Argumente deklarieren. Ein Argument wird dabei folgendermaßen deklariert:

```
[Attribute] [params | ref | out] Typ Name
```

Die Schlüsselworte `params`, `ref` und `out` beschreibe ich ab Seite 173. Für einfache Argumente reicht die Angabe des Typs und des Namens. Mehrere Argumente trennen Sie durch Kommata.

Wenn die Methode einen Typ zurückgibt, geben Sie diesen innerhalb der Methode durch die `return`-Anweisung zurück. `return` beendet außerdem die Ausführung der Methode.

Der folgende Quellcode deklariert eine Klasse zur Verwaltung von Mitarbeiterdaten. Eine Methode berechnet das Gehalt des Mitarbeiters und bekommt dazu zwei Argumente übergeben:

```
public class Mitarbeiter
{
   public string Vorname;
```

```
   public string Nachname;

   /* Methode zur Berechnung des Gehalts */
   public double GehaltBerechnen(double stunden,
      double bonus)
   {
      double gehalt;
      if (stunden <= 40)
         gehalt = stunden * 50;
      else
         gehalt = (40 * 50) + ((stunden - 40) * 75);
     gehalt += gehalt * bonus;
     return gehalt;
   }
}
```

Eine eigene Methode wird natürlich aufgerufen, wie jede andere Methode auch. Da die Methode nicht statisch ist, muss das Programm eine Instanz der Klasse erzeugen:

```
Mitarbeiter ma = new Mitarbeiter();
ma.Vorname = "Zaphod";
ma.Nachname = "Beeblebrox";

/* Aufruf der Methode */
double gehalt = ma.GehaltBerechnen(46, 0.1);

Console.WriteLine(ma.Vorname +
    " verdient mit 10% Bonus " +
    gehalt.ToString() + " Euro");}
```

4.6.2 Überladene Methoden

Wenn Sie Methoden überladen, deklarieren Sie mehrere Varianten der Methode. Die einzelnen Varianten müssen sich in den Typen der Argumente (in der so genannten *Signatur*) unterscheiden. Der Rückgabetyp kann unterschiedlich sein, wird aber nicht als Unterscheidungsmerkmal gewertet. Zwei Methoden mit derselben Signatur, aber unterschiedlichen Rückgabetypen sind nicht möglich. Der Compiler muss einfach die Chance haben, die einzelnen Varianten der Me-

thode über die Datentypen der beim Aufruf übergebenen Argumente zu identifizieren.

Das folgende Beispiel verdeutlicht dies anhand der Klasse zur Speicherung von Mitarbeiterdaten. Die Methode zur Berechnung des Gehalts wird mit einer zweiten Variante überladen, die das Gehalt ohne Bonus berechnet:

```
public class Mitarbeiter
{
   public string Vorname;
   public string Nachname;

   /* Eine Variante der GehaltBerechnen-Methode
    * berechnet das Gehalt des Mitarbeiters
    * ohne Bonus */
   public double GehaltBerechnen(double stunden)
   {
      if (stunden <= 40)
         return stunden * 50;
      else
         return (40 * 50) + ((stunden - 40) * 75);
   }

   /* Eine andere Variante der GehaltBerechnen-Methode
    * bekommt neben den Stunden noch einen Bonus in
    * Prozent übergeben */
   public double GehaltBerechnen(double stunden,
      double bonus)
   {
      double gehalt;
      if (stunden <= 40)
         gehalt = stunden * 50;
      else
         gehalt = (40 * 50) + ((stunden - 40) * 75);
     gehalt += gehalt * bonus;
     return gehalt;
   }
}
```

Bei der Anwendung des Objekts können Sie nun die eine oder die andere Methode zur Berechnung des Gehalts verwenden:

```
/* Einen Mitarbeiter erzeugen und initialisieren */
Mitarbeiter ma = new Mitarbeiter();
ma.Vorname = "Zaphod";
ma.Nachname = "Beeblebrox";

/* Aufruf der ersten Variante der Methode */
double gehalt = ma.GehaltBerechnen(46);

Console.WriteLine(ma.Vorname +
   " verdient normalerweise " +
   gehalt.ToString() + " Euro");

/* Aufruf der zweiten Variante der Methode */
gehalt = ma.GehaltBerechnen(46, 0.1);

Console.WriteLine(ma.Vorname +
   " verdient mit 10% Bonus " +
   gehalt.ToString() + " Euro");
```

4.6.3 ref- und out-Argumente

Wenn Sie ein Argument einer Methode nicht mit `ref` oder `out` deklarieren, ist die Übergabeart dieses Arguments »By Value«. Wenn Sie an diesem Argument eine Variable übergeben, erzeugt der Compiler innerhalb der Methode einen neuen Speicherbereich auf dem Stack und kopiert den Wert der übergebenen Variablen dort hinein. Wenn Sie innerhalb der Methode mit dem Argument (also mit dem Stack-Speicherbereich) arbeiten, wird die äußere Variable davon nicht beeinflusst:

```
class Demo
{
   public void ByValDemo(int i)
   {
      i++;
   }
}
```

```
class Start
{
   [STAThread]
   static void Main(string[] args)
   {
      Demo d = new Demo();
      int i = 10;
      d.ByValDemo(i);
      // Hier ist i immer noch 10
      Console.WriteLine(i);
   }
}
```

Wenn Sie das Argument allerdings mit dem Schlüsselwort `ref` deklarieren, übergibt der Compiler eine Referenz auf die Variable (die Übergabeart ist dann »By Reference«).

```
public void RefDemo(ref int i)
{
   i++;
}
```


Beim Aufruf der Methode müssen Sie die Variable dann auch mit `ref` kennzeichnen (Gott allein weiß, warum ...):

```
Demo d = new Demo();
int i = 10;
d.RefDemo(ref i);
// Hier ist i = 11
```

Über solche Argumente können Sie auch mehrere Werte zurückgeben, indem Sie mehrere `ref`-Argumente deklarieren. Ich sehe darin allerdings nicht viel Sinn, da Methoden auch Strukturen oder selbst entwickelte Objekte zurückgeben können (die dann die verschiedensten Eigenschaften besitzen können).

`out`-Argumente verhalten sich ähnlich. Der Unterschied zwischen `ref` und `out` ist, dass bei einem `ref`-Argument der Wert der Variablen beim Aufruf übergeben wird und in der Methode zur Verfügung steht. In einem `out`-Argument wird der Wert einer Variablen nicht übergeben. In-

nerhalb der Methode ist das Argument also uninitialisiert. Sie können den Wert der übergebenen Variablen nicht auslesen. Dafür können Sie auch uninitialisierte Variablen übergeben (was bei `ref` nicht möglich ist).

4.6.4 Übergeben von Referenztypen

In C# sollten Sie Referenztypen immer By Value übergeben. Das unterscheidet C# von vielen anderen Sprachen, bei denen diese Typen By Reference übergeben werden müssen. In C# kopiert der Compiler aber den Wert der Referenz in die lokale Argument-Variable. Der Wert der Referenz ist die Adresse des Objekts im Speicher. Sie müssen natürlich beachten, dass Sie das übergebene Objekt nach außen verändern, wenn Sie innerhalb der Methode Eigenschaften des Objekts beschreiben.

Sie können Referenztypen auch mit `ref` *übergeben. Dann verlangsamen Sie die Bearbeitung des Objekts innerhalb der Methode allerdings, weil der Compiler zwei Referenzen auflösen muss: die auf die Objekt-Variable und die auf das Objekt.*

4.6.5 Variable Argumente mit params

Über das `params`-Schlüsselwort können Sie Argumente deklarieren, an denen der Aufrufer eine beliebige Anzahl Werte übergeben kann. `params` wird dazu immer mit einer Array-Deklaration verwendet und muss als letztes Argument der Methode eingesetzt werden. Der Compiler erlaubt nur ein `params`-Argument pro Methode.

So können Sie z. B. eine Methode erzeugen, die eine beliebige Anzahl `int`-Werte summiert und die Summe zurückgibt:

```
class ParamsDemo
{
   int Sum(params int[] numbers)
   {
      int result = 0;
      foreach(int i in numbers)
         result += i;
      return result;
```

```
    }
}
```

Beim Aufruf übergeben Sie nun kein, ein oder mehrere Argumente, die Sie wie gewohnt durch Kommata trennen:

```
ParamsDemo p = new ParamsDemo();

Console.WriteLine(p.Sum());
Console.WriteLine(p.Sum(1,2,3,4,5,6,7,8,9));
```

Alternativ könnten Sie auch ein Array ohne `params` deklarieren. Dann muss der Aufrufer aber erst ein Array erzeugen und dieses dann übergeben. `params` erleichtert die Übergabe von Werten mit variabler Anzahl ernorm.

4.7 Konstruktoren und der Destruktor

Konstruktoren

Ein Konstruktor ist eine spezielle Methode, die automatisch aufgerufen wird, wenn das Objekt erzeugt wird. Auch wenn Sie keinen eigenen Konstruktor in die Klasse integrieren, besitzt diese immer einen Konstruktor: In Klassen, die keinen Konstruktor enthalten, fügt der Compiler automatisch einen parameterlosen Standardkonstruktor ein. Sie können eigene Konstruktoren in die Klasse implementieren, mit denen Sie ein Objekt direkt bei dessen Erzeugung initialisieren können. Bei der Personenklasse wäre es z. B. sinnvoll, einem Personenobjekt bei der Erzeugung gleich den Vornamen und den Nachnamen der Person zu übergeben.

In C# wird ein Konstruktor ähnlich einer Methode deklariert. Dabei wird allerdings kein Rückgabetyp angegeben. Außerdem muss der Name des Konstruktors derselbe sein, wie der der Klasse. Konstruktoren können wie Methoden überladen werden. Das folgende Beispiel demonstriert dies anhand einer Klasse zur Speicherung von Personendaten:

```
public class Person
{
   public string Vorname;
```

```
   public string Nachname;
   public string Ort;

   /* Dem ersten Konstruktor wird der Vorname und
    * der Nachname der Person übergeben */
   public Person(string vorname, string nachname)
   {
      this.Vorname = vorname;
      this.Nachname = nachname;
      this.Ort = "";
   }

   /* Dem zweiten Konstruktor wird zusätzlich
    * der Ort übergeben */
   public Person(string vorname, string nachname,
      string ort)
   {
      this.Vorname = vorname;
      this.Nachname = nachname;
      this.Ort = ort;
   }
}
```

Wenn Sie nun Objekte dieser Klasse erzeugen, müssen Sie einen der beiden Konstruktoren verwenden:

```
/* Eine Person mit dem ersten
 * Konstruktor erzeugen */
Person p1 = new Person("Donald", "Duck");

/* Eine Person mit dem zweiten
 * Konstruktor erzeugen */
 Person p2 = new Person("Fred Bogus", "Trumper",
    "New York");
```

Ein Konstruktor kann einen anderen Konstruktor der Klasse aufrufen. Der Aufruf muss vor der ersten Anweisung erfolgen. Deshalb müssen Sie den Aufruf mit einem Doppelpunkt getrennt an den Kopf des Konstruktors anhängen. Verwenden Sie dazu den `this`-Operator gefolgt von der Argumentliste. Der zweite Konstruktor der Personenklasse

kann z. B. den ersten aufrufen, um den Vornamen und den Nachnamen zu setzen:

```
public Person(string vorname, string nachname,
   string ort): this(vorname, nachname)
{
   this.Ort = ort;
}
```

Konstruktoren besitzen normalerweise die Sichtbarkeit `public`, können aber auch mit `private`, `protected` und `internal` deklariert werden. `public`-Konstruktoren können als einzige von außen verwendet werden, um ein Objekt dieser Klasse zu erzeugen.

Ein privater Konstruktor verhindert, dass ein Programm ein Objekt dieser Klasse mit Hilfe dieses Konstruktors erzeugt. Enthält die Klasse lediglich einen privaten Konstruktor, kann niemand (außer die Klasse selbst über eine statische Methode) von dieser Klasse Instanzen erzeugen. Üblicherweise wird ein solcher Konstruktor bei Klassen verwendet, die nur statische Elemente enthalten (siehe ab Seite 182), und von denen deshalb keine Instanzen erzeugt werden sollen.

`protected`-Konstruktoren verhalten sich ähnlich, können aber – im Gegensatz zu privaten Konstruktoren – von abgeleiteten Klassen in deren Konstruktoren aufgerufen werden.

Mit privaten und geschützten Konstruktoren können Sie einige spezielle Techniken programmieren. So können Sie z. B. eine Instanzierung einer Klasse nur über eine statische Methode der Klasse ermöglichen, damit Sie die Anzahl der erzeugten Instanzen kontrollieren können.

Ein `internal`-Konstruktor ermöglicht das Instanzieren eines Objekts innerhalb der Assemblierung, verhindert das Instanzieren aber von außerhalb. Diese Konstruktoren werden häufig in Klassenbibliotheken verwendet (Seite 220). In vielen Fällen ist es dort sinnvoll, Objekte einer Klasse nur von Methoden anderer Klassen erzeugen zu lassen und das Instanzieren von außen zu verhindern.

Konstruktoren werden nicht nur dazu verwendet, das Objekt zu initialisieren. Sie können in einem Konstruktor z. B. auch eine Datei oder eine Verbindung zu einer Datenbank öffnen, die Ihre Klasse benötigt.

Das Beispiel beim Destruktor im nächsten Abschnitt demonstriert dies.

Der Destruktor

Der Destruktor einer Klasse wird kurz bevor das Objekt vom Garbage Collector aus dem Speicher entfernt wird aufgerufen. Destruktoren sind normalerweise für Aufräumarbeiten gedacht, die beim Zerstören des Objekts auf jeden Fall ausgeführt werden müssen.

Der Begriff »Destruktor« ist für C#-Klassen eigentlich nicht korrekt, wird aber trotzdem des Öfteren (u. a. auch von Microsoft) verwendet. Programmiersprachen, bei denen die Objekte direkt nachdem keine Referenz mehr auf das Objekt zeigt, oder direkt nach dem Aufruf von speziellen Löschbefehlen zerstört werden, besitzen echte Destruktoren. Diese echten Destruktoren sind keine Methoden und werden nur implizit bei der Zerstörung des Objekts aufgerufen. Sprachen, bei denen ein Garbage Collector die Entsorgung der Objekte übernimmt, besitzen keine echten Destruktoren, sondern so genannte »Finalisierungsmethoden«. Eine Finalisierungsmethode ist im Prinzip eine ganz normale Methode, nur dass diese automatisch vom Garbage Collector aufgerufen wird, wenn das Objekt zerstört wird. Für die Praxis unterscheiden sich Destruktoren und Finalisierungsmethoden allerdings nur dadurch, dass der Aufruf der Finalisierungsmethode zu einem unbestimmten Zeitpunkt erfolgt.

Einen Destruktor deklarieren Sie ohne Modifizierer für die Sichtbarkeit und mit dem Namen der Klasse, dem allerdings eine Tilde (~) vorangestellt werden muss:

```
public class Demo
{
   ~Demo()
   {
      Console.WriteLine("Destruktor");
   }
}
```

C#-Destruktoren (bzw. Finalisierungsmethoden) sind eigentlich recht problematisch: Man weiß nie genau, wann der Garbage Collector das Objekt freigibt. Es kann sein, dass Ihr Programm gerade sehr beschäftigt ist und das Objekt erst eine halbe Stunde nach der Freigabe der Referenzen auf das Objekt aus dem Speicher entfernt wird. Microsoft beschreibt sogar, dass es möglich sein kann, dass der Destruktor nie aufgerufen wird. Deshalb sollten Sie nichts wirklich Wichtiges im Destruktor ausführen.

Wenn Sie im Konstruktor Ressourcen geöffnet haben, die möglichst schnell wieder freigegeben werden müssen (wie z. B. Datenbankverbindungen) können Sie diese also nicht einfach im Destruktor freigeben. Verwenden Sie dazu besser eine separate Methode. Microsoft empfiehlt, diese Methode »Close« zu nennen, wenn die Ressource (über eine andere Methode) wieder geöffnet werden kann und »Dispose«, wenn die Ressource damit endgültig geschlossen wird.

Das folgende Beispiel implementiert eine Logdatei-Klasse, die im Konstruktor eine Datei öffnet. Damit die Datei wieder ordnungsgemäß geschlossen werden kann, besitzt die Klasse eine Dispose-Methode:


```
public class Logdatei
{
   /* Eine private Variable für den StreamWriter,
    * der das Schreiben in die Datei ermöglicht */
   System.IO.StreamWriter sw;

   /* Im Konstruktor wird die Datei geöffnet */
   public Logdatei(string filename)
   {
      sw = new System.IO.StreamWriter(filename, true);
   }

   /* Eine Log-Methode wird zum Protokollieren
    * verwendet */
   public void Log(string info)
   {
```

```
      sw.WriteLine(DateTime.Now.ToString() + ": " +
         info);
   }

    /* Eine eigene Dispose-Methode wird verwendet,
     * um die Datei unabhängig vom Destruktor
     * zu schließen */
   public void Dispose()
   {
      sw.Close();
   }

   /* Im Destruktor wird die Datei
    * geschlossen, falls dies nicht bereits
    * über die Dispose-Methode geschehen ist */
   ~Logdatei()
   {
      sw.Close();
   }
}
```

Wird diese Klasse nun verwendet, wird schließlich die Dispose-Methode aufgerufen:

```
/* Ein Objekt der Klasse Logdatei erzeugen */
Logdatei l = new Logdatei(@"C:\Demolog.txt");

/* Und einen Text protokollieren */
l.Log(@"C# macht Spass.");

/* Datei über die Dispose-Methode schließen */
l.Dispose();
```

Die using-Anweisung

Über die `using`-Anweisung, die nichts mit der `using`-Direktive gemeinsam hat, können Sie festlegen, dass die Dispose-Methode eines Objekts direkt und automatisch nach der Verwendung des Objekts aufgerufen wird. Die Klasse muss dazu die `IDisposable`-Schnittstelle implementieren. Schnittstellen werden erst ab Seite 211 behandelt.

Ich wollte den Trick aber nicht verheimlichen. Ändern Sie die Deklaration der Klasse folgendermaßen ab:

```
public class Logdatei: IDisposable
```

Wenn Sie nun eine Instanz dieser Klasse in der using-Anweisung erzeugen und innerhalb der Anweisungsblöcke mit dieser Instanz arbeiten, ruft der Compiler nach dem Anweisungsblock automatisch die Dispose-Methode auf (die Bestandteil der `IDisposable`-Schnittstelle ist):

```
using(Logdatei log = new Logdatei(@"C:\Demolog.txt"))
{
   log.Log("using erleichtert die Arbeit");
}
/* Hier wird automatisch Dispose aufgerufen */
```

Das besonders geniale der `using`-Anweisung ist, dass die Dispose-Methode auch dann aufgerufen wird, wenn im `using`-Anweisungsblock eine Ausnahme erzeugt wird, die innerhalb des Blocks nicht behandelt wird. Ressourcen werden so also auf jeden Fall freigegeben.

4.8 Statische Klassenelemente

Normale Eigenschaften und Methoden sind immer einer Instanz einer Klasse zugeordnet. Manchmal besteht aber auch der Bedarf, dass eine Eigenschaft nur ein einziges Mal für alle Instanzen dieser Klasse gespeichert ist oder dass eine Eigenschaft oder Methode ohne Instanz der Klasse verwendet werden kann.

4.8.1 Statische Eigenschaften (Klasseneigenschaften)

Eine statische Eigenschaft ist nur einmal für die Klasse und nicht für jedes Objekt separat gespeichert. Damit können Sie Werte zwischen den einzelnen Instanzen der Klasse austauschen oder die globalen Variablen der strukturierten Programmierung simulieren.

In einer Kontoverwaltung ist z. B. der Kontostand jedem Kontoobjekt separat zugeordnet, der Zinssatz sollte aber für alle Kontoobjekte gemeinsam gespeichert sein. Diese Eigenschaft sollte statisch sein und wird einfach mit dem Modifizierer `static` deklariert:

```
public class Konto
{
   private int nummer;
   public double Stand;

   /* eine statische Eigenschaft für den Zinssatz */
  public static double Zinssatz;

   /* Der Konstruktor */
   public Konto(int nummer, double stand)
   {
      this.nummer = nummer;
      this.Stand = stand;
   }

   /* Methode zum Berechnen der Zinsen */
   public double ZinsenRechnen(int anzahlMonate)
   {
      return (Stand * Zinssatz) *
         (anzahlMonate / 12D);
   }
}
```

Alle Methoden (normale und statische) des Objekts können auf statische Eigenschaften zugreifen. Die *ZinsenRechnen*-Methode des Beispiels demonstriert dies. Eine statische Eigenschaft kann auch ohne Instanz einer Klasse gesetzt und geschrieben werden. Geben Sie dazu den Klassennamen als Präfix an:

```
/* Den Zinssatz der Konto-Objekte definieren */
Konto.Zinssatz = 0.025;

/* Einige Konto-Objekte erzeugen */
Konto konto1 = new Konto(1001, 1000);
Konto konto2 = new Konto(1002, 1200);

/* Den Zinsssatz der Account-Objekte ausgeben */
Console.WriteLine("Der aktuelle Zinssatz ist " +
   Konto.Zinssatz);
```

```
/* Zinsen für 6 Monate rechnen */
Console.WriteLine("Zinsen für Konto 1: " +
   konto1.ZinsenRechnen(6));
Console.WriteLine("Zinsen für Konto 2: " +
   konto2.ZinsenRechnen(6));
```

Globale Daten in statischen Eigenschaften

In manchen Fällen ist es sinnvoll oder notwendig, Daten so zu verwalten, dass diese global im gesamten Programm gelten. Eigentlich sollten solche Daten grundsätzlich (und besonders bei der OOP vermieden werden). Globale Daten beinhalten viele potenzielle Fehlerquellen, da jeder Programmteil auf diese Daten zugreifen und diese (eventuell fehlerhaft) verändern kann. Außerdem machen globale Daten ein Programm sehr undurchsichtig. Wenn Sie das Risiko aber eingehen wollen oder müssen, verwenden Sie dazu einfach eine Klasse mit lediglich statischen Eigenschaften:

```
public class Globals
{
   public static string DatabaseName;
   public static string UserName;
   public static string UserPassword;

   /* Ein privater Konstruktor verhindert
      eine Instanzierung dieser Klasse */
   private Globals()
   {
   }
}
```

Im Programm können Sie die »globalen Variablen« dieser Klasse dann über den Klassennamen referenzieren:

```
Globals.DatabaseName = "C:\Bestellungen.mdb"
```

4.8.2 Statische Methoden (Klassenmethoden)

Statische Methoden können – wie statische Eigenschaften – ebenfalls ohne Instanz der Klasse verwendet werden. Diese Methoden

verwenden Sie (sehr selten) für Operationen, die sich auf die Klasse und nicht auf einzelne Objekte beziehen. Über statische Methoden können Sie beispielsweise die statischen (privaten) Eigenschaften einer Klasse bearbeiten. Ein Beispiel dafür fehlt hier, weil diese Technik doch eher selten eingesetzt wird.

Wesentlich häufiger werden statische Methoden eingesetzt, um globale, allgemein anwendbare, Funktionen in das Projekt einzufügen. Das .NET-Framework nutzt solche Methoden sehr häufig. Die Klasse `Math` besteht z. B. fast ausschließlich aus statischen Methoden.

Wenn Sie z. B. in Ihren Projekten häufig verschiedene Maße und Gewichte in nationale Einheiten umrechnen müssen, können Sie dazu eine Klasse mit statischen Methoden verwenden:

```
/* Klasse mit statischen Methoden zur Maß- und
 * Gewichtseinheitsumrechnung */
public class UnitConversion
{
   public static double KmToMile(double Value)
   {
      return 0.6214 * Value;
   }

   public static double MileToKm(double Value)
   {
      return 1.609 * Value;
   }

   public static double LitreToUSGallon(double Value)
   {
      return Value * 0.264;
   }

   public static double USGallonToLitre(double Value)
   {
      return Value * 3.785;
   }

   /* Ein privater Konstruktor verhindert
```

```
      eine Instanzierung dieser Klasse */
   private UnitConversion()
   {
   }
}
```

Die Verwendung statischer Methoden für allgemeine Aufgaben ist wesentlich einfacher, als wenn Sie dazu immer wieder Objekte instanzieren müssten. Um eine Methode der Klasse *UnitConversion* zu verwenden, müssen Sie lediglich den Klassenamen angeben:

```
Console.WriteLine("1 km entspricht " +
   UnitConversion.KmToMile(1) + " Meilen");
Console.WriteLine("1 Liter entspricht " +
   UnitConversion.LitreToUSGallon(1) + " US-Gallonen");
```

Statische Methoden können nicht auf die normalen (Instanz-)Eigenschaften einer Klasse zugreifen. Das ist auch logisch, denn diese Eigenschaften werden erst im Speicher angelegt, wenn eine Instanz der Klasse erzeugt wird.

4.8.3 Statische Konstruktoren

Neben Eigenschaften und Methoden können auch Konstruktoren statisch deklariert werden. Ein statischer Konstruktor wird nur ein einziges Mal aufgerufen, nämlich dann, wenn die erste Instanz der Klasse erzeugt wird. Damit können Sie Initialisierungen vornehmen, die für alle Instanzen der Klasse gelten sollen. Deklarieren Sie diesen Konstruktor wie einen normalen Konstruktor, lediglich ohne Sichtbarkeits-Modifizierer und mit dem Schlüsselwort `static`.

Eine Kontoverwaltung könnte z. B. den aktuellen Zinssatz in einem statischen Konstruktor aus einer Datenbank auslesen (im Beispiel wird dies nur simuliert):

```
public class Konto
{
   public int Nummer;
   public double Stand;
```

```
   /* eine statische Eigenschaft für den Zinssatz */
   private static double Zinssatz;

   /* Ein statischer Konstruktor */
   static Konto()
   {
      /* Dieser Kontruktor ermittelt den
       * Zinssatz, wenn das erste Objekt dieser
       * Klasse instanziert wird */
      Zinssatz = 0.025;
   }

   /* Methode zum Berechnen der Zinsen */
   public double CalcInterest(int anzahlMonate)
   {
      return (Stand * Zinssatz) * (anzahlMonate / 12D);
   }
}
```

Ein statischer Konstruktor darf keinen Modifizierer für die Sichtbarkeit und keine Argumente besitzen. Neben einem statischen Konstruktor können Sie natürlich auch normale Konstruktoren deklarieren. Statische Destruktoren sind übrigens nicht möglich.

4.9 Verschachtelte Klassen

Manchmal ist es hilfreich, Klassen in andere Klassen zu verschachteln. Das ist z. B. immer dann der Fall, wenn eine Hilfsklasse nur von einer einzigen Klasse verwendet wird. Da dieses Feature in der Praxis wohl eher selten eingesetzt wird, finden Sie hier nur ein kurzes, abstraktes Beispiel. Das Beispiel, das ich ursprünglich entwickelt hatte, war einfach zu komplex. Sie finden dieses aber in den Buchbeispielen.

```
public class Outer
{
   /* Methoden und Eigenschaften der äußeren Klasse */
   public void Demo()
   {
```

```
      Console.WriteLine("Äußere Klasse");
      /* Instanz der inneren Klasse erzeugen und
         verwenden*/
      Inner i = new Inner();
      i.Demo();
   }

   /* Innere Klasse */
   public class Inner
   {
      public void Demo()
      {
         Console.WriteLine("Innere Klasse");
      }
   }
}
```

Eine innere Klasse ist immer noch eine Klasse. Der wichtigste Unterschied zu normalen Klassen ist, dass innere Klassen von außen nur über den Namen der äußeren Klasse referenziert werden können (wenn Sie die innere Klasse `public` deklariert haben):

```
/* Instanz der äußeren Klasse erzeugen */
Outer o = new Outer();
o.Demo();
/* Instanz der inneren Klasse erzeugen */
Outer.Inner i = new Outer.Inner();
i.Demo();
```

Was im Beispiel nicht dargestellt ist, ist die Tatsache, dass innere Klassen auf alle statischen Elemente der äußeren Klasse zugreifen können, unabhängig von deren Sichtbarkeit. Der Zugriff auf nichtstatische Elemente ist allerdings nicht möglich.

4.10 Vererbung und Polymorphismus

Die Vererbung ist neben der Kapselung ein weiteres, sehr wichtiges Grundkonzept der OOP. Über die Vererbung können Sie neue Klassen erzeugen, die alle Eigenschaften und Methoden einer Basisklasse er-

ben. So können Sie sehr einfach die Funktionalität einer bereits vorhandenen Klasse erben, erweitern und/oder neu definieren. Da Sie auch von Klassen erben können, die in einer Klassenbibliothek vorliegen (also kompiliert sind), besitzen Sie damit eine geniale Möglichkeit der Wiederverwendung bereits geschriebener Programme. Wenn Sie z. B. eine Auflistung (Collection) eigener Objekte programmieren wollen, können Sie Ihre Auflistungsklasse von der .NET-Klasse `DictionaryBase` ableiten. Diese Klasse enthält bereits die für eine Auflistung grundlegenden Methoden und Eigenschaften und muss nur noch um einige Methoden erweitert werden (wie ich es in Kapitel 5 beschreibe).

Die Vererbung wird in der Praxis für eigene Klassen nur relativ selten eingesetzt. Ein Grund dafür ist, dass selten wirklicher Bedarf für Vererbung besteht. Ein anderer Grund ist, dass Programme, die Vererbung sehr intensiv nutzen, meist recht unübersichtlich und fehleranfällig werden. Da die Klassen des .NET-Framework aber intensiv Vererbung einsetzen und Sie manchmal Ihre eigenen Klassen von .NET-Basisklassen ableiten müssen, sollten Sie die Vererbung schon beherrschen.

Bei der Vererbung spricht man häufig auch von ableiten: Eine Klasse wird von einer Basisklasse abgeleitet. Die Basisklasse wird auch als Superklasse, die abgeleitete Klasse auch als Subklasse bezeichnet.

Wenn Sie Klassen von anderen Klassen ableiten, kommt automatisch auch Polymorphismus ins Spiel. Diesen Begriff und den Umgang damit erläutere ich aber erst ab Seite 195.

4.10.1 Ableiten von Klassen

Wenn Sie eine Klasse von einer anderen Klasse (der Basisklasse) ableiten wollen, geben Sie die Basisklasse einfach mit einem Doppelpunkt getrennt hinter der Subklasse an.

```
public class Girokonto: Konto
{
}
```

Sie können nur eine Klasse als Basisklasse angeben, C# unterstützt (wie die meisten OOP-Sprachen) lediglich die Einfachvererbung[2].

Diese Deklaration reicht bereits aus, um eine neue Klasse zu erzeugen, die alle Elemente der Basisklasse erbt. Die Sichtbarkeit der Klassenelemente wird dabei nicht verändert. Alle öffentlichen Elemente sind in der abgeleiteten Klasse ebenfalls öffentlich, alle privaten bleiben privat.

Sinn macht die Vererbung allerdings erst dann, wenn Sie die abgeleitete Klasse erweitern oder geerbte Methoden neu definieren bzw. überschreiben.

4.10.2 Erweitern von Klassen

Das Erweitern von abgeleiteten Klassen ist prinzipiell einfach. In der abgeleiteten Klasse deklarieren Sie dazu lediglich neue Eigenschaften und Methoden. Das Beispiel, dass ich hier entwerfe, deklariert zuerst eine Basisklasse *Konto* mit den Eigenschaften *Nummer* und *Stand*, einem Konstruktor, dem diese Grunddaten übergeben werden und der Methode *Einzahlen*. Die Klasse *Girokonto* wird davon abgeleitet und um eine Eigenschaft *Dispobetrag* und eine Methode zum Abheben erweitert. Die Abheben-Methode überprüft, ob die Auszahlung des Betrags unter Berücksichtigung des Dispobetrags möglich ist. Zunächst beschreibe ich aber noch zwei wichtige Probleme beim Erweitern und deren Lösung.

Konstruktoren werden nicht öffentlich vererbt

Wenn Sie eine Klasse von einer anderen Klasse ableiten, erbt die abgeleitete Klasse wohl die Konstruktoren der Basisklasse. Diese werden aber auf jeden Fall verborgen. Wenn Sie keine eigenen Konstruktoren in die Klasse einfügen, fügt der Compiler einen parameterlosen Standardkonstruktor ein. C# verbirgt geerbte Elemente, wenn diese in der neuen Klasse auch nur einmal neu eingefügt werden. In einer abgeleiteten Klasse sind die geerbten Konstruktoren nach außen hin also nicht sichtbar. Die Klasse *Girokonto* soll aber wie die Klasse

2. C++ unterstützt auch die Mehrfachvererbung, bei der eine Subklasse von mehreren Superklassen erben kann.

Konto einen Konstruktor besitzen, dem die Kontonummer und der Stand des Kontos übergeben werden kann. Dieser Konstruktor muss dann neu implementiert werden. Sie können dabei aber den geerbten Konstruktor aufrufen. Dazu hängen Sie das Schlüsselwort `base` (das Zugriff auf die geerbten Elemente der Basisklasse gibt) gefolgt von der Parameterliste, durch einen Doppelpunkt getrennt an den Konstruktorkopf an.

```
public Girokonto(int number, double stand)
    :base(number, stand)
```

Zugriff auf eigentlich private Elemente über protected

Ein anderes Problem ist in vielen Fällen, dass die abgeleitete Klasse auf eigentlich private Elemente der Basisklasse zugreifen muss. Das Girokonto muss z. B. in seiner *Abheben*-Methode auf die geerbte Eigenschaft *stand* zugreifen, auf die von außen kein Zugriff möglich sein soll. Solche Eigenschaften können in der Basisklasse nicht privat deklariert werden, da eine abgeleitete Klasse keinen Zugriff auf die privaten Elemente einer Basisklasse besitzt. Verwenden Sie für solche Elemente dann in der Basisklasse den Modifizierer `protected`. Geschützte Elemente gelten nach außen wie private, werden aber so vererbt, dass die abgeleitete Klasse Zugriff darauf hat.

Das Beispiel

Das folgende Listing implementiert die beiden Beispielklassen *Konto* und *Girokonto* unter Berücksichtigung der genannten Probleme:

```
/* Eine Klasse für ein einfaches Konto */
public class Konto
{
   public readonly int Nummer;

   /* Die Eigenschaft stand wird als protected
    * gekennzeichnet, damit das Girokonto darauf
    * zugreifen kann */
   protected double stand;

   /* öffentliche Nur-Lese-Eigenschaft für den
    * Kontostand */
```

```
   public double Stand {get {return stand;}}

   /* Methode zum Einzahlen */
   public void Einzahlen(double betrag)
   {
      stand += betrag;
   }

   /* Der Konstruktor */
   public Konto(int nummer, double stand)
   {
      this.Nummer = nummer;
      this.stand = stand;
   }
}

/* Die Klasse Girokonto (Girokonto) wird von Konto
 * abgeleitet und um einen Dispobetrag und eine
 * Methode zum Abheben erweitert */
public class Girokonto: Konto
{
   /* Der Dispobetrag */
   public double Dispobetrag = -1000;

   /* C# erzeugt einen parameterlosen
    * Defaultkonstruktor in jede neue Klasse, der alle
    * geerbten Konstruktoren nach außen verbirgt,
    * deshalb muss ein Konstruktor mit Parametern in
    * der neuen Klasse noch einmal deklariert werden */
   public Girokonto(int nummer, double stand)
      :base(nummer, stand)
   {
   }

   /* Methode zum Abheben */
   public void Abheben(double betrag)
   {
      if (stand - betrag >= Dispobetrag)
```

```
            stand -= betrag;
        else
            /* Wenn das Abheben nicht möglich ist,
             * wird eine Ausnahme erzeugt */
            throw new Exception("Abheben nicht " +
               "möglich. Der Dispobetrag würde " +
               "unterschritten werden.");
    }
}
```

Wenn Sie die Klassen nun verwenden, können Sie auf ein Konto und ein Girokonto einzahlen, von einem Girokonto abheben und von beiden Konten die Nummer und den Betrag auslesen:

```
Konto konto1 = new Konto(1001, 100);
konto1.Einzahlen(1000);

Girokonto konto2 = new Girokonto(1002, 1000);
konto2.Einzahlen(1000);
konto2.Abheben(500);

Console.WriteLine("Konto Nummer " + konto1.Nummer +
   ": " + konto1.Stand);
Console.WriteLine("Konto Nummer " + konto2.Nummer +
   ": " + konto2.Stand);
```

4.10.3 Neudefinieren von Methoden

Eine abgeleitete Klasse kann nicht nur neue Methoden und Eigenschaften hinzufügen, sondern auch geerbte Methoden mit einer neuen Implementierung definieren. Eine Klasse könnte z. B. die Daten von Mitarbeitern speichern und das Gehalt berechnen:

```
public class Mitarbeiter
{
   public string Vorname;
   public string Nachname;

   /* Der Konstruktor */
   public Mitarbeiter(string vorname, string nachname)
   {
```

```
      this.Vorname = vorname;
      this.Nachname = nachname;
   }

   /* Eine Methode zur Berechnung des Gehalts */
   public double GehaltBerechnen(int stunden)
   {
      return (stunden * 100);
   }
}
```

Eine davon abgeleitete Klasse speichert die Daten von Chefs und berechnet das Gehalt anders:

```
public class Chef: Mitarbeiter
{
   /* Der Konstruktor */
   public Chef(string vorname, string nachname)
      :base(vorname, nachname)
   {

   }

   /* Die Methode zur Berechnung des Gehalts
    * wird neu definiert */
   public double GehaltBerechnen(int stunden)
   {
      return stunden * 500;
   }
}
```

Bei dieser Variante meldet der Compiler allerdings eine Warnung, dass die Methode *GehaltBerechnen* die geerbte Methode ausblendet (verbirgt). Die Warnung fällt weg, wenn Sie den `new`-Modifizierer verwenden, der aussagt, dass diese Methode eine neue Version der geerbten Methode ist:

```
public new double GehaltBerechnen(int stunden)
```

Diese Art des Neudefinierens wird auch als »Verbergen« bezeichnet. Die neue Methode verbirgt die geerbte Methode, sodass diese von außen nicht mehr aufgerufen werden kann.

Beachten Sie, dass das einfache Neudefinieren von Methoden immer dann zu Problemen führt, wenn Sie die Klasse in polymorphen Situationen verwenden. Polymorphismus, dieses Problem und dessen Lösung werden im folgenden Abschnitt beschrieben.

C# überschreibt Methoden nach deren Namen, nicht nach der Signatur. Wenn Sie in einer abgeleiteten Klasse nur eine Methode neu implementieren, die in verschiedenen überladenen Varianten in der Basisklasse vorhanden ist, werden alle Varianten der Methode in der abgeleiteten Klasse nach außen hin verborgen. Wenn Sie auch die einzelnen Varianten der Methode in der abgeleiteten Klasse zur Verfügung stellen wollen, müssen Sie alle diese Varianten neu implementieren. In den neuen Methoden können Sie allerdings natürlich die geerbten Methoden aufrufen.

4.10.4 Polymorphismus und virtuelle Methoden

Polymorphismus (Vielgestaltigkeit) bedeutet, dass ein Objekt in mehreren Gestalten vorkommen kann. Eigentlich ist diese Aussage nicht ganz korrekt, denn ein Objekt gehört immer einer Klasse an und besitzt deswegen auch nur eine Gestalt, nämlich die, die durch die Klasse festgelegt wird. Aber eine Klasse kann ja von einer Basisklasse abgeleitet werden und erbt damit alle Eigenschaften und Methoden dieser Klasse. In der abgeleiteten Klasse ist es möglich, geerbte Methoden mit einer neuen Programmierung zu überschreiben. Objekte der abgeleiteten Klasse verhalten sich anders als Objekte der Basisklasse, wenn eine überschriebene Methode aufgerufen wird. Das ist Polymorphismus.

Polymorphismus kann nicht nur durch Vererbung erreicht werden, sondern auch über Schnittstellen. Schnittstellen werden ab Seite 211 beschrieben.

Die Anwendung von Polymorphismus bedeutet, dass ein Programmteil, der eine Referenz auf ein Objekt der Basisklasse besitzt, über diese Referenz auch ein Objekt einer davon abgeleiteten Klasse bearbeiten kann. Über die Vererbung besitzt ein Objekt der abgeleiteten Klasse ja auf jeden Fall die Eigenschaften und Methoden der Basisklasse, und zwar in genau derselben Form (d. h. mit derselben Signatur). Das Ganze ist am Anfang ziemlich schwierig zu verstehen, wenn Sie Polymorphismus allerdings einmal am praktischen Beispiel angewendet haben, ist dieser nicht mehr ganz so mysteriös.

Sie wissen jetzt, was Polymorphismus ist, aber vielleicht nicht, wann Sie diesen einsetzen. Da in diesem Buch kein Platz für eine ausführliche Erläuterung ist, verweise ich auf meinen Online-Artikel »OOP-Grundlagen«. Dort wird Polymorphismus sehr ausführlich beschrieben.

In einer Mitarbeiterverwaltung könnten z. B. die Klassen für Mitarbeiter und Chefs verwendet werden, um alle Mitarbeiter in einem Array zu verwalten:


```
Mitarbeiter[] ma = new Mitarbeiter[2];
ma[0] = new Mitarbeiter("Ford", "Prefect");
ma[1] = new Chef("Zaphod", "Beeblebrox");

Console.WriteLine(ma[0].Vorname +
   " verdient in 160 Stunden " +
   ma[0].GehaltBerechnen(160) + " Euro");

Console.WriteLine(ma[1].Vorname +
   " verdient in 160 Stunden " +
   ma[1].GehaltBerechnen(160) + " Euro");
```

Dem Programm ist es möglich, über eine Referenz auf die Basisklasse *Mitarbeiter* auch Objekte der Klasse *Chef* zu bearbeiten. Die Klasse *Chef* hat ja alle Elemente der Klasse *Mitarbeiter* geerbt. Der Compiler kann sich darauf verlassen, dass die Klasse *Chef* die Eigenschaften und Methoden der Klasse *Mitarbeiter* besitzt.

Falls die Klassen allerdings so aussehen, wie in Abschnitt 4.10.3 (Seite 193), entsteht dabei ein großes Problem:

Wenn Sie Ihre Klassen polymorph verwenden wollen, sollten Sie auf das einfache Neudefinieren von Methoden verzichten. Der Compiler verwendet dann nämlich immer die Version der Methode, die zu dem Typ der Referenz passt (und eben nicht die Version, die zum Typ des Objekts passt). Wollten Sie z. B. alle Mitarbeiter in einem Array verwalten und das Gehalt dieser Mitarbeiter berechnen, kommt es mit der vorliegenden Lösung zu dem Problem, dass ein Chef genau so viel verdient wie ein einfacher Mitarbeiter.

Um dieses Problem zu verhindern, müssen Sie die Methode in der Basisklasse als virtuelle Methode kennzeichnen und in der abgeleiteten Klasse mit dem Schlüsselwort `override` überschreiben. Virtuelle Methoden werden in einer so genannten »VTable« oder »VMT« (Virtual Method Table) verwaltet, womit der Compiler immer die korrekte, überschriebene Methode der abgeleiteten Klasse verwendet, unabhängig vom Typ der Referenz:

```
/* Eine Klasse für Mitarbeiterdaten */
public class Mitarbeiter
{
   public string Vorname;
   public string Nachname;

   /* Der Konstruktor */
   public Mitarbeiter(string vorname, string nachname)
   {
      this.Vorname = vorname;
      this.Nachname = nachname;
   }
   /* Eine Methode zur Berechnung des Gehalts.
    * Die Methode wird virtuell deklariert,
    * damit abgeleitete Klassen diese Methode
    * ohne Probleme überschreiben können und
    * Objekte der abgeleiteten Klasse ohne
    * Probleme mit Referenzen auf die Basiklasse
    * verwendet werden können */
  public virtual double GehaltBerechnen(int stunden)
   {
```

```
      return (stunden * 100);
   }
}

/* Eine Klasse für die Daten des Chefs*/
public class Chef: Mitarbeiter
{
   /* Der Konstruktor */
   public Chef(string vorname, string nachname)
      :base(vorname, nachname)
   {
   }
   /* Die Methode zur Berechnung des Gehalts
    * wird überschrieben */
  public override double GehaltBerechnen(int stunden)
   {
      return stunden * 500;
   }
}
```

Wenn Sie nun im Programm mit einer Referenz auf die Mitarbeiter-Klasse einen Chef verwalten und das Gehalt berechnen, ruft der Compiler immer die Methode auf, die in der Klasse deklariert ist. Nun erhält ein Chef auch das, was er verdient.

Virtuelle Methoden bleiben in abgeleiteten Klassen grundsätzlich immer virtuell und müssen dort mit dem Modifizierer `override` überschrieben werden.

Die Zusammenhänge, die zum Verständnis virtueller Methoden notwendig sind, sind recht komplex und können aus Platzgründen hier nicht dargestellt werden. In meinem Online-Artikel »OOP-Grundlagen« finden Sie eine ausführliche Erläuterung. Merken sollten Sie sich, dass Sie immer dann, wenn Sie auch nur ahnen, dass später Instanzen von abgeleiteten Klassen über Referenzen vom Typ der Basisklasse verwendet werden, alle Methoden der Basisklasse virtuell deklarieren sollten. Der Aufruf virtueller Methoden benötigt zwar etwas mehr Zeit als der normaler Methoden (wegen des Umwegs über die VTable), bringt Ihnen aber die Sicherheit, dass alles funktioniert.

4.10.5 Vererbung von Konstruktoren und Destruktoren

Entgegen den Aussagen mancher Publikationen[3] (u. a. auch der C#-Sprachspezifikation) werden Konstruktoren und Destruktoren – meiner Ansicht nach – vererbt, aber nicht so, dass diese in abgeleiteten Klassen nach außen sichtbar sind. Im Prinzip verhält sich ein Konstruktor bzw. Destruktor bei der Vererbung wie ein geschütztes (protected) Klassenelement. Sie müssen also alle Konstruktoren der Basisklasse neu implementieren, wenn Sie diese in der abgeleiteten Klasse zur Verfügung haben wollen.

Enthält ein Konstruktor keinen Aufruf eines geerbten Konstruktors, fügt der Compiler automatisch einen Aufruf des parameterlosen Standardkonstruktors als erste Anweisung in den Programmcode ein. Das ist auch in einem vom Compiler implizit hinzugefügten Standardkonstruktor der Fall. Besitzt die Basisklasse keinen parameterlosen Konstruktor (z. B. weil Sie einen oder mehrere Konstruktoren mit Argumenten in die Klasse eingefügt haben) führt dies zu dem Fehler »Keine Überladung für die Methode '*Basisklasse*' benötigt 'o' Argumente«. In solchen Fällen müssen Sie in einem Konstruktor der abgeleiteten Klasse also einen geerbten Konstruktor explizit aufrufen. Wenn Sie selbst einen geerbten Konstruktor aufrufen wollen, verwenden Sie den `base`-Operator, gefolgt von der eventuellen Argumentliste. Diese Anweisung muss allerdings direkt hinter dem Kopf des Konstruktors, durch einen Doppelpunkt getrennt angegeben werden. Der folgende Quellcode demonstriert dies an einem einfachen Beispiel. Eine Basisklassen enthält lediglich einen Konstruktor mit Argument und folglich keinen parameterlosen Konstruktor:

```
class Person
{
   private string name;

   /* Konstruktor */
   public Person(string name)
```

3. In vielen Publikationen ist zu lesen, dass Konstruktoren und Destruktoren bei den meisten OOP-Sprachen nicht vererbt werden. Ich habe das nie verstanden, denn eine abgeleitete Klasse kann alle Konstruktoren der Basisklasse aufrufen. Wie soll das gehen, wenn diese nicht vererbt werden?

```
    {
        this.name = name;
    }
}
```

Eine davon abgeleitete Klasse besitzt ebenfalls einen Konstruktor mit Argument:

```
class Mitarbeiter: Person
{
    /* Konstruktor mit Aufruf des geerbten
     * Konstruktors */
    public Mitarbeiter(string name):base(name)
    {
    }
}
```

Würden Sie den Aufruf des geerbten Konstruktors weglassen, so würde der Compiler an dieser Stelle den parameterlosen Konstruktor aufrufen wollen, der aber nicht existiert. Sie erhalten dann den Fehler, dass keine Überladung der Methode *Person* o Argumente besitzt.

In einem Destruktor fügt der Compiler einen Aufruf des geerbten Destruktors immer automatisch als letzte Anweisung ein. Ein expliziter Aufruf ist nicht nötig und auch nicht möglich.

Damit ist gewährleistet, dass in abgeleiteten Klassen die Konstruktoren und Destruktoren der Basisklassen immer in der korrekten Reihenfolge aufgerufen werden.

4.10.6 Zugriff auf geerbte Elemente

Die Methoden der abgeleiteten Klasse können auf alle Methoden und Eigenschaften der Basisklasse zugreifen, die nicht privat deklariert sind. Sofern in der Klasse keine gleichnamigen Elemente mit gleicher Signatur existieren, kann dazu einfach der Name verwendet werden. Existieren in der abgeleiteten Klasse gleichnamige Elemente mit gleicher Signatur, können Sie das Schlüsselwort `base` verwenden, das expliziten Zugriff auf die geerbten Elemente gibt. Der besseren Übersicht wegen sollten Sie dieses Schlüsselwort immer verwenden, wenn Sie auf geerbte Elemente zugreifen.

In einer Kontoverwaltung könnte die *Abheben*-Methode z. B. bereits in der Basisklasse *Konto* definiert sein, in der abgeleiteten Klasse *Girokonto* überschrieben werden und dort auf die geerbte Methode zugreifen. Der Konstruktor der abgeleiteten Klasse ruft den Basiskonstruktor auf:

```
/* Eine Klasse für ein einfaches Konto */
public class Konto
{
   public readonly int Nummer;
   private double stand;
   public double Stand {get {return stand;}}

   /* Methode zum Einzahlen */
   public virtual void Einzahlen(double betrag)
   {
      stand += betrag;
   }

   /* Methode zum Abheben */
   public virtual void Abheben(double betrag)
   {
      stand -= betrag;
   }

   /* Der Konstruktor */
   public Konto(int Nummer, double stand)
   {
      this.Nummer = Nummer ;
      this.stand  = stand ;
   }
}

/* Die Klasse Girokonto wird von Konto abgeleitet
 * und um einen Dispobetrag und um eine Methode zum
 * Abheben erweitert */
public class Girokonto: Konto
{
   /* Der Dispobetrag */
```

```
   public double Dispobetrag = -1000;

   /* Der Konstruktor */
   public Girokonto(int Nummer, double stand)
      :base(Nummer, stand) {}

   /* Die Methode zum Abheben wird überschrieben */
   public override void Abheben(double betrag)
   {
      if (Stand  - betrag >= Dispobetrag)
         /* Aufruf der geerbten Methode */
         base.Abheben(betrag);
      else
         throw new Exception("Abheben nicht " +
            "möglich. Der Dispobetrag würde " +
            "unterschritten werden.");
   }
}
```

In diesem Beispiel wurden die Methoden der Basisklasse virtuell deklariert, weil anzunehmen ist, dass diese Methoden in abgeleiteten Klassen überschrieben werden und Objekte dieser Klassen in polymorphen Situationen eingesetzt werden.

4.11 Abstrakte Klassen und Methoden

Abstrakte Klassen sind im Allgemeinen Klassen, von denen nicht sinnvoll Objekte instanziert und verwendet werden können. In einer Kontoverwaltung könnte die Klasse *Konto* z. B. nur als Basisklasse für die Klassen *Girokonto* und *Sparkonto* gedacht sein. Um zu verhindern, dass der Programmierer ein Objekt der Klasse *Konto* erzeugt, können Sie diese mit dem Modifizierer `abstract` kennzeichnen:

```
public abstract class Konto
```

Den Versuch, ein Objekt einer abstrakten Klasse zu erzeugen, quittiert der Compiler mit dem Fehler, dass von abstrakten Klassen keine Instanzen erzeugt werden können.

```
Konto konto1 = new Konto(1001,2500); // Fehler
```

Ein Objekt einer abgeleiteten Klasse kann allerdings erzeugt werden:

```
Girokonto konto2 = new Girokonto(1002, 2500);
```

In abstrakten Klassen macht es häufig keinen Sinn, Methoden zu implementieren. Das ist oft der Fall, wenn die abgeleiteten Klassen polymorph, d. h. mit einer Referenz auf die Basisklasse, verwendet werden sollen. Die Basismethoden müssen dann zwar in der Basisklasse vorhanden sein, werden aber nicht implementiert. C# stellt Ihnen für solche Methoden ebenfalls den Modifizierer `abstract` zur Verfügung. Abstrakte Methode dürfen keinen Rumpf enthalten und sind implizit virtuell, müssen also in abgeleiteten Klassen mit dem Schlüsselwort `override` überschrieben werden:

```
/* Abstrakte Basisklasse für die Kontoverwaltung */
public abstract class Konto
{
   public readonly int Nummer;
   protected double stand;
   public double Stand {get {return stand;}}

   /* Methode zum Einzahlen */
   public virtual void Einzahlen(double betrag)
   {
      stand += betrag;
   }

   /* Die Methode zum Abheben soll in abgeleiteten
    * Klassen überschrieben werden und wird deswegen
    * abstrakt deklariert */
   public abstract void Abheben(double betrag);

   /* Der Konstruktor */
   public Konto(int Nummer, double stand)
   {
      this.Nummer = Nummer;
      this.stand = stand;
   }
}
```

```
/* Die Klasse Girokonto wird von Konto
 * abgeleitet und definiert die abstrakt
 * geerbte Methode Abheben */
public class Girokonto: Konto
{
   public double Dispobetrag = -1000;

   /* Der Konstruktor */
   public Girokonto(int Nummer, double stand)
      :base(Nummer, stand) { }

   /* Die Methode zum Abheben wird überschrieben */
   public override void Abheben(double betrag)
   {
      if (stand - betrag >= Dispobetrag)
         /* Aufruf der geerbten Methode */
         base.stand -= betrag;
      else
         throw new Exception("Abheben nicht " +
            "möglich. Dispobetrag würde " +
            "unterschritten werden.");
   }
}
/* Die Klasse Sparkonto (Sparkonto) wird von Konto
 * abgeleitet und definiert die abstrakte Methode
 * Abheben mit einer anderen Funktion als bei der
 * Klasse Girokonto */
public class Sparkonto: Konto
{
   /* Der Konstruktor */
   public Sparkonto(int Nummer, double stand)
      :base(Nummer, stand) { }

   /* Die Methode zum Abheben wird überschrieben */
  public override void Abheben(double betrag)
   {
      if (stand - betrag >= 0)
         /* Aufruf der geerbten Methode */
         base.stand -= betrag;
      else
         throw new Exception("Abheben nicht " +
```

```
            "möglich, da ein negativer " +
            "Betrag resultieren würde");
    }
}
```

4.12 Versiegelte Klassen

Sie können auf einfache Weise verhindern, dass von Ihren Klassen andere Klassen abgeleitet werden können. Das macht immer dann Sinn, wenn Sie Ihre Klassen in Klassenbibliotheken (Seite 220) an andere Programmierer verteilen und sicherstellen wollen, dass kein Programmierer in abgeleiteten Klassen die Funktionalität Ihrer Klasse »verbiegt«, also u. U. Fehler einbaut. In einer Kontoverwaltung ist es daher sinnvoll, die Klassen für das Girokonto und das Sparkonto zu versiegeln. Ansonsten könnte ein Programmierer mutwillig oder versehentlich beispielsweise die *Abheben*-Methode so überschreiben, dass diese ein unbegrenztes Abheben ermöglicht und Schaden anrichtet.

Verwenden Sie zum Versiegeln einfach den Modifizierer `sealed` (»versiegelt«).

```
public sealed class Girokonto: Konto
```

Ein Versuch, von dieser Klasse eine Subklasse abzuleiten, resultiert dann in einem Compilerfehler.

Logischerweise können Sie die Modifizierer `abstract` und `sealed` nicht gemeinsam verwenden, da der eine eine Ableitung erzwingt und der andere eine Ableitung verhindert.

4.13 Operatoren für Klassen

Wenn bei den Objekten Ihrer Klassen die Anwendung der Standardoperatoren Sinn macht, können Sie diese für die Verwendung mit Instanzen Ihrer Klassen überladen. Bei einer Klasse für XY-Koordinaten (Punkte) wäre es beispielsweise vorteilhaft, die Standardoperatoren in der Klasse zu überladen. Stattdessen könnten Sie natürlich auch Methoden verwenden, aber die Anwendung eines Operators ist in vielen Fällen intuitiver als die einer Methode:

```
Point p1 = new Point(3, 3);
Point p2 = new Point(4, 4);
Point p3 = new Point(0, 0);
/* Intuitive Anwendung eines Operators */
p3 = p1 + p2;
/* Weniger intuitive Anwendung einer Methode */
p3 = p1.Add(p2);
```

Überladen können Sie die unären Operatoren,

```
+  -  !  ~  ++  -
```

die binären Operatoren,

```
+  -  *  /  %  &  |  ^  <<  >>  ==  !=  >  <  >=  <=
```

die Literale `true` und `false` und die Operatoren für Konvertierungen.

Das Überladen der Operatoren `true` *und* `false`*, das sehr selten benötigt wird, wenn Sie ermöglich wollen, dass Instanzen Ihrer Klassen mit* `true` *und* `false` *verglichen werden können, wird im separaten Online-Artikel »Überladen der Operatoren true und false« beschrieben.*

Unäre und binäre Operatoren

Unäre Operatoren überladen Sie nach dem folgenden Schema:

```
public static Ergebnistyp operator Operator
   (Klassentyp lhs)
{
   return Ergebnis;
}
```

Bei binären Operatoren wird dem Operator der linke und der rechte Operand übergeben:

```
public static Ergebnistyp operator Operator
   (Datentyp lhs, Datentyp rhs)
{
   return Ergebnis;
}
```

lhs steht für den linken, *rhs* für den rechten Operanden. Für verschiedene Vergleiche können Sie mehrere Varianten des Operators mit verschiedenen Datentypen überladen. Die wohl meist eingesetzte

Variante arbeitet mit zwei Argumenten vom Klassentyp. Daneben können Sie aber z. B. auch eine Operation mit einem Objekt dieser Klasse und einem anderen Datentyp implementieren.

Vergleichsoperatoren geben den Datentyp `bool` zurück, arithmetische Operatoren den Klassentyp. Operatoren, die logische Paare bilden, müssen gemeinsam überladen werden. Dazu gehören die Paare == und !=, < und > und <= und >=. Die zusammengesetzten Zuweisungsoperatoren +=, -=, *= etc. können nicht überladen werden, da diese immer in eine einfache Operation und Zuweisung zerlegt werden. Es reicht für diese Operatoren also aus, wenn Sie die Operatoren == und +, -, * etc. überladen.

Die folgende Klasse *Point* überlädt einige der Standardoperatoren:

```
public class Point
{
   public double X;
   public double Y;

   /* Der Konstruktor */
   public Point(double x, double y)
   {
      this.X = x;
      this.Y = y;
   }

   /* Die ToString-Methode wird überschrieben */
   public override string ToString()
   {
      return String.Format("{0}, {1}", X, Y);
   }

   /* Operator == */
   public static bool operator == (Point rhs,
      Point lhs)
   {
      return ((rhs.X == lhs.X) & (rhs.Y == lhs.Y));
   }
```

```
   /* Operator != */
   public static bool operator != (Point rhs,
      Point lhs)
   {
      return ((rhs.X != lhs.X) | (rhs.Y != lhs.Y));
   }

   /* Operator + */
   public static Point operator + (Point rhs,
      Point lhs)
   {
      return new Point(rhs.X + lhs.X, rhs.Y + lhs.Y);
   }

   /* Operator - */
   public static Point operator - (Point rhs,
      Point lhs)
   {
      return new Point(rhs.X - lhs.X, rhs.Y - lhs.Y);
   }

   /* Operator ++ */
   public static Point operator ++ (Point rhs)
   {
      return new Point(rhs.X + 1, rhs.Y + 1);
   }

   /* Operator -- */
   public static Point operator -- (Point rhs)
   {
      return new Point(rhs.X - 1, rhs.Y - 1);
   }
}
```


Bei der Verwendung der Klasse werden diese Operatoren nun intuitiv eingesetzt:

```
public class Start
{
```

```
    [STAThread]
    public static void Main(string[] args)
    {
        /* Einige Punkte erzeugen */
        Point p1 = new Point(2, 2);
        Point p2 = new Point(2, 2);
        Point p3 = new Point(3, 2);
        Point p4;

        /* Den ==-Operator verwenden */
        if (p1 == p2)
            Console.WriteLine("p1 ist gleich p2");
        else
            Console.WriteLine("p1 ist ungleich p2");

        /* Den !=-Operator verwenden */
        if (p1 != p3)
            Console.WriteLine("p1 ist ungleich p3");
        else
            Console.WriteLine("p1 ist gleich p3");

        /* Den +-Operator verwenden */
        p4 = p1 + p2;
        Console.WriteLine(p4.ToString());

        /* Den ++-Operator verwenden */
        p2++;
        Console.WriteLine(p2.ToString());

        /* Den automatisch erzeugten Operator +=
         * verwenden (die Klasse implementiert die
         * Operatoren* == und +, woraus sich der +=-
         * Operator zusammensetzt) */
        p2 += p1;
        Console.WriteLine(p2.ToString());
    }
}
```

Operatoren für Konvertierungen

Wenn Sie in Ihrer Klasse Konvertierungen unterstützen wollen, können Sie Operatoren für implizite und explizite Konvertierungen einfügen. Implizite Konvertierungen sollten unterstützt werden, wenn bei der Konvertierung keine Informationen verloren gehen können. Ein Objekt einer Klasse, die Musiknoten speichert, könnte z. B. implizit nach `double` konvertiert werden, weil ein Double-Wert ohne Probleme die Frequenzwerte aller Noten speichern kann:

```
/* Eine Musiknote, die 10 Töne vom Kammerton
 * A (440 Hz) entfernt ist */
Note note = new Note(10);
/* Implizite Konvertierung nach Double (in Hertz) */
double hertz = note;
```

Immer dann, wenn bei einer Konvertierung Informationen verloren gehen, sollte eine explizite Konvertierung unterstützt werden. Ein `double`-Wert (in Hertz) kann z. B. nur ungenau in eine Note umgewandelt werden:

```
double hertz = 500; // nächste Frequenz ist 494 Hz
Note n = (Note)hertz; // = Ton H = 2 Töne Abstand
```

Operatoren für implizite und explizite Konvertierungen werden wie im folgenden Beispiel in die Klasse eingefügt:

```
public class Note
{
    /* Die Eigenschaft Position speichert den Abstand
    * zwischen dem Kammerton A und dieser Note */
    public int Position;

    public Note(int position)
    {
        this.Position = position;
    }

    /* Operator zum impliziten Konvertieren einer
     * Note nach double */
  public static implicit operator double(Note n)
```

```
    {
        return 440 * Math.Pow(2,
        (double)n.Position / 12);
    }

    /* Operator zum expliziten Konvertieren eines
     * double-Werts in eine Note. Ein double-Wert
     * kann nur annähernd in eine Note konvertiert
     * werden. */
  public static explicit operator Note(double x)
    {
        return new Note((int)(0.5 + 12 *
        (Math.Log(x / 440) / Math.Log(2))));
    }
}
```

Eine implizite Konvertierung kann auch explizit verwendet werden:

```
Console.WriteLine("Eine Note, die " +
   note.Position.ToString() +  " Halbtöne von A " +
   "entfernt ist, besitzt die Frequenz " +
   ((double)note).ToString() + " Hertz.");
```

4.14 Schnittstellen

Schnittstellen (engl.: Interfaces) sind ein wichtiges Konzept der modernen OOP. Als Schnittstelle werden bei der OOP im Allgemeinen die öffentlichen (Public-)Methoden und Eigenschaften einer Klasse bezeichnet. Jede Klasse besitzt damit mindestens eine Schnittstelle. In modernen Programmiersprachen können Klassen jedoch auch mehrere Schnittstellen besitzen. Jede dieser Schnittstellen besitzt einen eigenen Satz an Eigenschaften und Methoden. Einer Klasse eine oder mehrere zusätzliche Schnittstellen hinzuzufügen ist sehr einfach (schwieriger ist wohl das Verstehen des Sinns von Schnittstellen): Sie deklarieren eine Schnittstelle ähnlich einer Klasse mit ausschließlich abstrakten Methoden und geben die Schnittstelle wie bei der Vererbung bei der Deklaration einer Klasse an. Eine typische Schnittstellendeklaration könnte so aussehen:

```
Interface IPrintable
{
   void Print();
}
```

Eine Schnittstelle darf Eigenschaften, Methoden, Ereignisse und Indizierer enthalten, aber keine Implementierung der Methoden oder Eigenschaften (falls diese mit Zugriffsmethoden arbeiten). Die Implementierung erfolgt in der Klasse, die diese Schnittstelle einbindet. Eine Klasse kann mehrere Schnittstellen einbinden:

```
/* Schnittstelle, die für den Ausdruck verwendet
 * werden soll */
public interface IPrintable
{
   void Print();
}

/* Schnittstelle, die für die Ausgabe von
 * Informationen zum Objekt verwendet
 * werden soll */
public interface IInfo
{
   void MessageBox();
}

/* Die Klasse Person implementiert beide
 * Schnittstellen */
public class Person: IPrintable, IInfo
{
   public string Vorname;
   public string Nachname;
   public string Ort;

   public Person(string vorname, string nachname,
      string ort)
   {
      this.Vorname = vorname;
      this.Nachname = nachname;
```

```
      this.Ort = ort;
   }

   /* Implementierung der Methoden der
    * IPrintable-Schnittstelle */
  public void Print()
   {
      Console.WriteLine(Vorname + " " + Nachname +
         "\n" + Ort);
   }
   /* Implementierung der Methoden der
    * IInfo-Schnittstelle */
  public void MessageBox()
   {
      System.Windows.Forms.MessageBox.Show(Vorname +
         " " + Nachname + "\n" + Ort);
   }
}
/* Die Klasse Konto implementiert nur die
 * IPrintable-Schnittstelle */
public class Konto: IPrintable
{
   public int Number;
   public double Stand;
   public Konto(int number, double stand)
   {
      this.Number = number;
      this.Stand = stand;
   }

   /* Implementierung der Methoden der
    * IPrintable-Schnittstelle */
  public void Print()
   {
      Console.WriteLine("Konto Nummer " + Number +
         ": " + Stand + " Euro");
   }
}
```

Die Grundidee von Schnittstellen ist die Implementierung von Polymorphismus, ohne eine Klasse von einer bestimmten anderen Klasse ableiten zu müssen. Eine Klasse, die eine oder mehrere Schnittstellen implementiert, kann an eine Referenz auf eine dieser Schnittstellen (!) übergeben werden. Das Programm kann über die Referenz alle Eigenschaften und Methoden der Schnittstelle bearbeiten.

```
public class Start
{

    /* Eine Methode erwartet eine Referenz vom
     * Typ der Schnittstelle IPrintable */
    private static void Print(IPrintable ip)
    {
        /* Alle Elemente der Schnittstelle können
         * verwendet werden */
        ip.Print();
    }

   /* Eine Methode erwartet eine Referenz vom
    * Typ der Schnittstelle IInfo */
    private static void Info(IInfo ii)
    {
        /* Alle Elemente der Schnittstelle können
         * verwendet werden */
        ii.MessageBox();
    }

    public static int Main(string[] args)
    {
        /* Eine Person erzeugen */
        Person p = new Person("Smilla", "Jaspersen",
           "Kopenhagen");

        /* Die private Methode aufrufen, die eine
         * Referenz vom Typ der Schnittstelle
         * IPrintable erwartet */
        Print(p);
```

```
        /* Die private Methode aufrufen, die eine
         * Referenz vom Typ der Schnittstelle
         * IInfo erwartet */
        Info(p);

        /* Ein Konto erzeugen */
        Konto konto = new Konto(1001, 2560);

        /* Die private Methode aufrufen, die eine
        * Referenz vom Typ der Schnittstelle
        * IPrintable erwartet */
        Print(konto);
    }
}
```

Die zwei wesentlichen Vorteile gegenüber dem Polymorphismus über Vererbung (Seite 195) liegen darin, dass der Polymorphismus über Schnittstellen auch dann implementiert werden kann, wenn die Basisklasse nicht veränderbar ist (was z. B. beim Vererben von .NET-Framework-Klassen der Fall ist) und dass damit auch Mehrfach-Polymorphismus erreicht wird.

In den Klassen des .NET-Framework wird Polymorphismus über Schnittstellen recht häufig verwendet. Wenn Sie z. B. Objekte in einem Array speichern und dieses Array über `Array.Sort` sortieren, ruft die `Sort`-Methode für alle Objekte im Array die `CompareTo`-Methode der `IComparable`-Schnittstelle auf um zwei Objekte miteinander zu vergleichen. Die Klassen für die zu speichernden Objekte müssen also diese Schnittstelle implementieren. Im Abschnitt »Wichtige Methoden, Operatoren und Schnittstellen für eigene Klassen« ab Seite 219 finden Sie ein Beispiel dazu.

Entwurfsrichtlinien und Schnittstellen in Kombination mit Vererbung

Wenn Sie Polymorphismus erreichen wollen, ist die Entscheidung zwischen Vererbung und Schnittstellen manchmal nicht einfach. Dies gilt besonders im Hinblick auf abstrakte Basisklassen (die viel Ähnlichkeit mit Schnittstellen besitzen).

Um zu klären, wann Vererbung und wann Schnittstellen eingesetzt werden, können Sie sich vorstellen, dass Sie über Vererbung eine »Ist ein(e)«-Beziehung zwischen Subklassen und Superklassen aufbauen. Ein Kreis, ein Rechteck und ein Punkt *ist* z. B. *ein* Grafikobjekt. Hier bietet sich die Vererbung von einer (abstrakten) Basisklasse *Grafikobjekt* an. Mit Schnittstellen dagegen erreichen Sie, dass eine Klasse lediglich »eine Fähigkeit besitzt«. Das Beispiel auf den vorhergehenden Seiten zeigt z. B. die Klasse *Person* und die Klasse *Konto*, von denen man nicht sagen kann, dass diese »ein bestimmtes Basisobjekt sind«, aber, dass diese *die Fähigkeit besitzen*, ihre Daten auszudrucken. Hier wird sinnvollerweise eine Schnittstelle eingesetzt.

Manchmal bietet sich aber auch eine Kombination von Vererbung mit Schnittstellen an. Ein Kreis und ein Rechteck, die beide Grafikobjekte sind, *besitzen* daneben auch *die Fähigkeit*, skaliert zu werden, ein Punkt dagegen nicht:

```
/* Abstrakte Basisklasse für grafische Objekte */
public abstract class GraficObject
{
   /* Die Eigenschaften X, Y und Color besitzen alle
    * Grafikobjekte */
   public double X;
   public double Y;
   public int Color;
}

/* Schnittstelle zum Skalieren, die einige der
 * von GraficObject abgeleiteten Klassen
 * implementieren */
interface IScalable
{
   void Scale(double factor);
}


/* Die Klasse Point implementiert die
 * Schnittstelle nicht */
public class Point: GraficObject
```

```
{
   /* Der Konstruktor */
   public Point(double x, double y, int color)
   {
      X = x;
      Y = y;
      Color = color;
   }
}

/* Die Klasse Circle implementiert die
 * Schnittstelle */
public class Circle: GraficObject, IScalable
{
   /* Zusätzliche Eigenschaft */
   public double Radius;

   /* Der Konstruktor */
   public Circle(double x, double y, double radius,
      int Color)
   {
      X = x;
      Y = y;
      Radius = radius;
      Color = color;
   }

   /* Implementierung der Schnittstelle */
   public void Scale(double factor)
   {
      Radius *= factor;
   }
}

/* Die Klasse Rectangle implementiert die
 * Schnittstelle ebenfalls */
public class Rectangle: GraficObject, IScalable
{
```

```
   /* Zusätzliche Eigenschaften */
   public double Width;
   public double Height;

   /* Der Konstruktor */
   public Rectangle(double x, double y, double width,
      double height, int Color)
   {
      X = x;
      Y = y;
      Height = height;
      Width = width;
      Color = color;
   }

   /* Implementierung der Schnittstelle */
   public void Scale(double factor)
   {
      Height *= factor;
      Width *= factor;
   }
}
```


Wenn Sie nun Objekte dieser Klassen mit Referenzen auf die Basisklasse speichern,

```
GraficObject[] graficObjects = new GraficObject[5];
graficObjects[0] = new Point(10, 20, 100);
graficObjects[1] = new Rectangle(20, 30, 100, 90, 4);
graficObjects[2] = new Point(22, 33, 100);
graficObjects[3] = new Circle(300, 200, 100, 100);
graficObjects[4] = new Rectangle(200, 80, 220, 440, 2);
```

müssen Sie bei der Verwendung dieser Objekte überprüfen, ob eine Schnittstelle implementiert wird, wenn Sie deren Elemente verwenden wollen.

Abfragen, ob ein Objekt eine Schnittstelle implementiert

Besonders dann, wenn Sie mit einer Referenz auf eine Basisklasse Objekte abgeleiteter Klassen verwalten (»Polymorphismus über Ver-

erbung«), und nur bestimmte dieser Objekte eine bestimmte Schnittstelle implementieren, müssen Sie häufig herausfinden, ob ein Objekt eine bestimmte Schnittstelle implementiert, bevor Sie Elemente dieser Schnittstelle verwenden können. Verwenden Sie dazu den `Is`-Operator. Der folgende Quellcode geht die Grafikobjekte des vorigen Beispiels durch und skaliert die skalierbaren Objekte:

```
/* Array durchgehen und die skalierbaren Objekte
 * skalieren */
foreach (GraficObject g in graficObjects)
{
   /* Überprüfen, ob das aktuelle Grafikobjekt
    * die Schnittstelle IScalable implementiert */
   if (g is IScalable)
   {
      /* Referenz auf die Schnittstelle holen */
      IScalable scalable = (IScalable)g;
      /* Grafikobjekt über die Schnittselle
       * skalieren */
      scalable.Scale(0.5);
   }
}
```

4.15 Wichtige Methoden, Operatoren und Schnittstellen für eigene Klassen

Wenn Sie eigene Klassen entwickeln, sollten Sie einige Methoden, Operatoren und Schnittstellen in diese Klassen integrieren, damit die Objekte später möglichst universell einsetzbar sind. Zunächst sollten Sie die geerbte `ToString`-Methode überschreiben. `ToString` wird u. a. von Steuerelementen aufgerufen, wenn Sie ein Array oder eine Auflistung von Objekten an das Steuerelement binden (womit dieses dann einfach für alle gespeicherten Objekte einen Eintrag anzeigt). Datenbindung wird zwar erst in Kapitel 8 behandelt, als Richtlinie für eigene Klassen sollten Sie aber jetzt schon beachten, dass Sie dazu `ToString` sinnvoll implementieren sollten. Falls Ihre Klasse direkt von `object` abgeleitet ist, würde `ToString` ansonsten nur den Klassennamen ausgeben.

Wenn Objekte Ihrer Klasse in einem Array oder in einer der vielen Auflistungen (siehe Kapitel 5) aufgelistet werden, sollten Sie zudem noch die von `object` geerbte `Equals`-Methode überschreiben und die `IComparable`-Schnittstelle implementieren. Die `Equals`-Methode wird u. a. von der `IndexOf`-Methode eines Arrays bzw. einer Auflistung aufgerufen, die `CompareTo`-Methode von `IComparable` wird von der `Sort`- und der `BinarySearch`-Methode eines Arrays oder einer Auflistung verwendet. Implementieren Sie diese Methoden nicht, funktionieren die genannten Methoden eines Arrays bzw. einer Auflistung einfach nicht korrekt. Die `CompareTo`-Methode bekommt ein zu vergleichendes Objekt übergeben und muss die folgenden Werte zurückgeben: -1, wenn diese Instanz kleiner ist als die übergebene, 0, wenn beide Instanzen gleich groß sind und 1, wenn diese Instanz größer ist als die übergebene.

Da Sie `Equals` implementieren, sollten Sie zur Vervollständigung zumindest noch die == und !=-Operatoren implementieren. Andere Operatoren, wie >, < etc. stehen Ihnen natürlich frei.

In den Buchbeispielen zu Kapitel 4 finden Sie im Ordner »Wichtige Elemente bei eigenen Klassen« ein Projekt mit einer Klasse, die alles Genannte implementiert sowie deren Anwendung.

4.16 Klassenbibliotheken

Bevor ich auf erweiterte Techniken wie Delegates und Ereignisse zu sprechen komme, beschreibe ich hier kurz, wie Sie Klassenbibliotheken erzeugen und verwenden. Erweiterte Techniken machen häufig nämlich nur Sinn, wenn diese in Klassenbibliotheken verwendet werden.

Klassenbibliotheken entwickeln

Klassenbibliotheken sind in .NET sehr einfach zu erzeugen und anzuwenden: Wenn Sie eine Quellcodedatei erzeugen, die nur Klassen enthält und keinen Einsprungpunkt für eine Anwendung, können Sie diese in eine Klassenbibliothek kompilieren. Das folgende Beispiel zeigt eine solche Datei, die (aus Vereinfachungsgründen) nur eine einzelne Klasse enthält:

```
using System;

namespace Nitty_Gritty.Samples.Libraries
{
   /* Eine Klasse zur Speicherung von Personendaten */
   public class Person
   {
      public string Vorname;
      public string Nachname;
      public Person(string vorname, string nachname)
      {
         Vorname = vorname;
         Nachname = nachname;
      }
   }
}
```

Achten Sie darauf, dass Sie einen sinnvollen Namensraum einstellen. Wenn Sie die Klassen der Assemblierung in anderen Projekten verwenden, müssen Sie diesen Namensraum angeben.

Um eine Klassenbibliothek in Visual Studio zu erstellen, erzeugen Sie ein neues Projekt von Typ KLASSENBIBLIOTHEK. Die notwendigen Compileroptionen sind in diesem Projekt bereits eingestellt.

Klassenbibliotheken in Visual Studio testen

Um eine Klassenbibliothek in Visual Studio zu testen, müssen Sie diese nicht erst kompilieren und in einem anderen Projekt referenzieren. Sie können in Visual Studio einfach auch zwei Projekte in einer Projektmappe öffnen (jetzt bekommt die Projektmappe Sinn). Ein Projekt erzeugt die Klassenbibliothek, ein anderes enthält eine Testanwendung. Über das Menü DATEI / PROJEKT HINZUFÜGEN können Sie der Projektmappe neue und existierende Projekte hinzufügen.

In dem Testprojekt legen Sie dann eine Referenz auf das andere Projekt an. Wählen Sie dazu den Eintrag REFERENZ HINZUFÜGEN aus dem Kontextmenü des REFERENZEN-Eintrags im Projekt-Explorer. Über das Register PROJEKTE des erscheinenden Dialogs können Sie eine Referenz auf das Klassenbibliothek-Projekt erstellen.

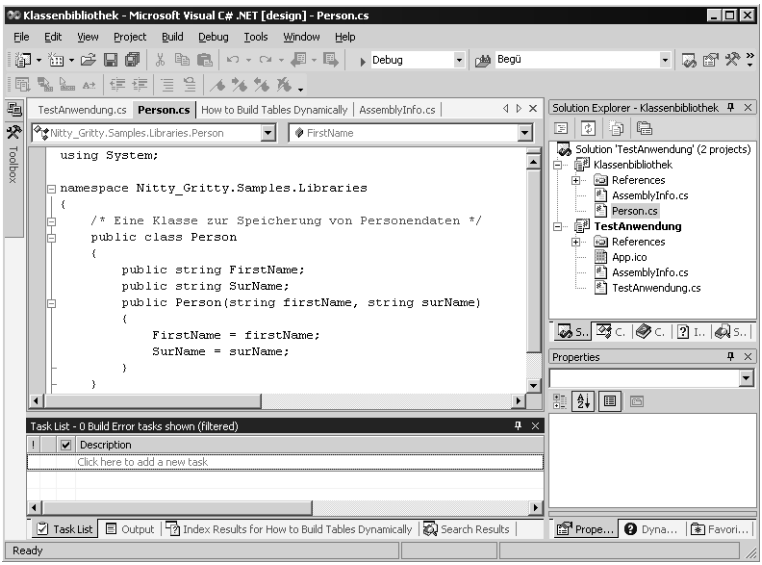

Bild 4.1: Visual Studio (in der englischen Version) mit einem Klassenbliblio-theks- und einem Konsolenanwendungs-Projekt

Achten Sie dann noch darauf, dass das Testprojekt als Startprojekt eingestellt ist. Sie erkennen dies daran, dass der Projektname im Projektmappen-Explorer fett formatiert ist. Bei Bedarf stellen Sie das Testprojekt dann über das Kontextmenü des Projekteintrags im Projektmappen-Explorer als Startprojekt ein.

Wenn Sie diese Projektmappe nun kompilieren, erzeugt der Compiler – neben der eigentlichen Anwendung – für die Klassenbibliothek eine Assemblierung mit der Endung *.dll*. Sie können die Klassen der Bibliothek aber auf diese Art sehr gut testen und debuggen.

Klassenbibliotheken mit dem Kommandozeilencompiler kompilieren

Wenn Sie den Kommandozeilencompiler verwenden, müssen Sie beim Kompilieren die Option `/target:library` angeben.

Klassenbibliotheken referenzieren

Fertige Klassenbibliotheken können Sie in anderen Projekten verwenden. Sie können die Assemblierung an beliebiger Stelle speichern. So können Sie Ihre eigenen Assemblierungen z. B. in einem eigenen Ordner gemeinsam ablegen, damit Sie diese bei Bedarf schnell wiederfinden.

Eine Assemblierung, die eine Klassenbibliothek darstellt, müssen Sie referenzieren, damit Sie deren Typen verwenden können. In Visual Studio machen Sie dies – wie beim Testen – über das Kontextmenü des `Verweise`-Eintrags im Projekt-Explorer. Fügen Sie einen Verweis auf die Assemblierung hinzu, indem Sie die Datei über das Register PROJEKTE suchen und hinzufügen. Wenn Sie den Kommandozeilencompiler verwenden, referenzieren Sie externe Assemblierungen über die `/r`-Option. Wenn die Assemblierung in einem anderen Ordner als dem Ordner der Anwendung gespeichert ist, müssen Sie diesen Ordner über die `/lib`-Option angeben:

```
csc /r:MyLib.dll /lib:C:\MyAssemblies TestApp.cs
```

Wenn eine Anwendung ausgeführt wird, muss die CLR alle Referenzen auflösen können. Dazu sucht die CLR die richtigen Versionen der referenzierten Assemblierungen. Der Vorgang ist ziemlich kompliziert. Unter anderem werden dabei Anwendungs- und Maschinen-Konfigurationen und Kultur-Informationen verwendet. Sie finden eine Beschreibung dieses Vorgangs – der den Rahmen dieses Buchs sprengen würde –, wenn Sie bei *search.microsoft.com/us/dev* nach »How the Runtime Locates Assemblies« suchen.

Wenn Sie zunächst kulturspezifische Assemblierungen und Anwendungskonfigurationen außer Acht lassen, können Sie davon ausgehen, dass die CLR die referenzierten Assemblierungen zuerst im Ordner der Anwendung und dann im globalen Assemblierungen-Cache sucht.

Wenn Sie eine Anwendung, die eine Klassenbibliothek verwendet, in Visual Studio kompilieren, kopiert Visual Studio die Assemblierung in den Ordner *bin/debug* oder *bin/release*. Die Assemblierung befindet sich also im Ordner der Anwendung. Um die Anwendung auf ei-

nem anderen Rechner zu installieren, müssen Sie lediglich alle Assemblierungen aus diesem Ordner kopieren.

Wenn Sie mit dem Kommandozeilencompiler kompilieren, müssen Sie die referenzierten Assemblierungen zum Testen und Ausführen der Anwendung selbst in den Ordner des Projekts kopieren. Alternativ können Sie aber auch eine Konfigurationsdatei für die Anwendung verwenden, wie ich es in Kapitel 9 beschreibe.

4.17 Delegates und Ereignisse

Was ein Ereignis ist, wissen Sie ja bereits. Ein Objekt generiert in bestimmten Situationen Ereignisse, die vom Anwender des Objekts in einer Ereignisbehandlungsmethode abgefangen werden können. Dieser Abschnitt beschreibt nun, wie Sie Ereignisse und die ähnlichen Delegates deklarieren und verwenden.

Ich habe hier leider nicht den Platz, die Anwendung von Delegates und Ereignissen ausführlich zu erläutern. In den Buchbeispielen finden Sie drei Projekte, die ein typisches Problem über eine Schnittstelle, über einen Delegate und über ein Ereignis lösen. Diese Projekte bringen vielleicht ein wenig Licht ins Dunkel.

4.17.1 Delegates

Delegates sind eigentlich nichts anderes als Methodenzeiger, über die eine Methode andere Methoden aufrufen kann. Diese Methoden müssen jedoch nicht zur Kompilierzeit bekannt sein, sondern können vom Programmierer in der Laufzeit instanziert und an die Referenz übergeben werden. Besonders interessant sind Delegates in Klassenbibliotheken, wenn eine Methode einer Klasse in bestimmten Situationen eine andere Methode aufrufen soll, die vom Anwender der Klassenbibliothek programmiert werden soll.

Das folgende Beispiel ist bewusst einfach gehalten. Eine Klasse in einer Klassenbibliothek soll irgend etwas ausführen (was, ist jetzt unwichtig) und dabei in bestimmten Situationen Informationen über einen Zustand liefern (loggen). Was mit den Informationen passieren soll, soll der Anwender der Klasse entscheiden. Die Klasse deklariert

dazu einen Delegate, der genau wie eine abstrakte Methode deklariert wird, lediglich mit dem Modifizierer `delegate`:

```
using System;

namespace Library
{
   /* Deklaration eines Delegate zum Loggen */
  public delegate void Logger(string message);

   /* Klasse, deren ProcessJob-Methode
    * den Delegate aufruft */
   public class WorkerClass
   {
      public void ProcessJobs(Logger logger)
      {
         // irgend etwas ausführen
         System.Threading.Thread.Sleep(1000);
         /* Loggen */
     if (logger != null)
            logger("Done the first job");
         // etwas anderes ausführen
         System.Threading.Thread.Sleep(1000);
         /* Loggen */
      if (logger != null)
            logger("Done the second job");
         // etc.
      }
   }
}
```

Eine Anwendung, die diese Klassenbibliothek verwendet, kann nun eine Methode erzeugen, die die Signatur der Logger-Methode besitzt. Die Anwendung muss dann nur noch eine neue Instanz des Delegates erzeugen und die Log-Methode an diese Instanz übergeben:

```
using System;
using Library;

namespace App
```

```
{

   class Start
   {
      /* Methode für den Delegate */
    static void logHandler(string message)
      {
         Console.WriteLine(message);
      }

      [STAThread]
      static void Main(string[] args)
      {
         /* Instanz der Klasse WorkerClass erzeugen */
         WorkerClass worker = new WorkerClass();
         /* Die Methode aufrufen, die ihre Arbeit
          * protokollieren soll. Dabei wird eine
          * Instanz der Log-Methode übergeben */
         worker.ProcessJobs(new Logger(logHandler));

         Console.WriteLine("Ready");
      }
   }
}
```

Die *ProcessJobs*-Methode ruft nun die *logHandler*-Methode in der Anwendung auf, wenn protokolliert werden soll.

Wenn die Anwendung keine Methode deklariert, kann diese einfach den Wert `null` an die *ProcessJobs*-Methode übergeben. Diese überprüft ja, ob der Delegate auf eine Methoden-Instanz zeigt und ruft die Methode nur dann auf.

4.17.2 Multicast-Delegates

Wenn ein Delegate einen `void`-Rückgabetyp besitzt, kann dieser mehrere Methodenreferenzen verwalten und damit auch mehrere Methoden aufrufen. Sinn macht das, wenn die Delegate-Referenz eine Eigenschaft der Klasse ist:

```
using System;

namespace Library
{
   /* Deklaration eines Delegate zum Loggen */
   public delegate void Logger(string message);

   /* Klasse, deren ProcessJob-Methode
    * den Delegate aufruft */
   public class WorkerClass
   {
      public Logger Logger;

      public void ProcessJobs()
      {
         // irgendetwas ausführen
         System.Threading.Thread.Sleep(1000);
         /* Loggen */
         if (Logger != null)
            Logger("Done the first job");
         // etwas anderes ausführen
         System.Threading.Thread.Sleep(1000);
         /* Loggen */
         if (Logger != null)
            Logger("Done the second job");
         // etc.
      }
   }
}
```

Die Anwendung kann nun über den Operator += mehrere Methoden zuweisen:

```
using System;
using Library;

namespace App
{
   class Start
```

```
{
   /* Methode für den Delegate */
 static void logHandler1(string message)
   {
      Console.WriteLine("Log Handler 1: " +
         message);
   }

   static void logHandler2(string message)
   {
      Console.WriteLine("Log Handler 2: " +
         message);
   }


   [STAThread]
   static void Main(string[] args)
   {
      /* Instanz der Klasse WorkerClass erzeugen */
      WorkerClass worker = new WorkerClass();
      /* Instanz des einen Log-Handlers übergeben */
  worker.Logger += new Logger(logHandler1);
      /* Instanz des anderen Handlers übergeben */
   worker.Logger += new Logger(logHandler2);
      /* Die Methode aufrufen, die ihre Arbeit
       * protokollieren soll.*/
      worker.ProcessJobs();

      Console.WriteLine("Ready");
   }
  }
}
```

Die *ProcessJobs*-Methode ruft nun beide Methoden auf.

4.17.3 Delegates im Vergleich zu Schnittstellen

Ein Problem, dass Sie mit Delegates lösen können, kann auch mit Schnittstellen gelöst werden. Der Unterschied für den Benutzer der

Klasse ist allerdings, dass er dann eine Klasse deklarieren muss, die die geforderte Schnittstelle implementiert. Der Vorteil von Delegates ist, dass diese eleganter sind und dass die erzeugten Methoden im Gültigkeitsbereich der anwendenden Klasse liegen, die das Objekt mit den Delegates verwendet. In den Buchbeispielen finden Sie Projekte, die ein solches Problem einmal mit einer Schnittstelle und dann mit Delegates und Ereignissen lösen.

4.17.4 Ereignisse

Ereignisse verhalten sich ähnlich wie Multicast-Delegates. Ein kleiner Unterschied ist allerdings vorhanden: Bei Delegates kann ein Programm, das Zugriff auf ein Objekt hat, die Delegate-Methode nachträglich entfernen oder eine neue Delegate-Methode setzen:

```
worker.Logger = null;
...
worker.Logger = new Logger(otherLogHandler);
```

In größeren Projekten kann das zum Problem werden. Die zuvor gesetzten Delegate-Methoden werden dann schließlich nicht mehr aufgerufen.

Ereignisse fügen dem Ganzen eine gewisse Sicherheit hinzu, indem diese nur die Operatoren += und -= erlauben. So kann eine Anwendung Ereignisbehandlungsmethoden nur hinzufügen oder explizit einzeln entfernen.

Für Ereignisse ist unter .NET zudem ein Standard definiert: Delegates sollten immer zwei Argumente besitzen. Im ersten Argument übergibt ein Ereignis eine `object`-Referenz auf das Objekt, das das Ereignis generiert, im zweiten ein Objekt der Klasse `EventArgs` oder einer davon abgeleiteten Klasse. Außerdem sollten Sie sich an die Namensrichtlinien halten (siehe Kapitel 3).

Ereignisse werden über eine Eigenschaft der Klasse verwaltet, so wie ich dies beim Multicast-Delegate-Beispiel auch schon programmiert habe. Ereignisse unterscheiden sich von solchen Delegates zunächst nur durch die Deklaration der Eigenschaft mit dem `event`-Schlüsselwort. Zusätzlich müssen Sie aber eine Klasse deklarieren, die von `EventArgs` abgeleitet ist und die eventuelle, neue Ereignisargumente

als Eigenschaft besitzt, wenn Sie spezielle Argumente an den Aufrufer übergeben wollen.

```
using System;

namespace Library
{
   /* Deklaration einer Klasse für die Ereignis-
    * Argumente */
   public class LogEventArgs: EventArgs
   {
      /* Zusätzlich Eigenschaft */
      public string Message;
      /* Konstruktor */
      public LogEventArgs(string message)
      {
         Message = message;
      }
      /* Überschreiben der ToString-Methode */
      public override string ToString()
      {
         return Message;
      }
   }

   /* Deklaration eines Delegate zum Loggen */
   public delegate void LogHandler(object sender,
      LogEventArgs e);

   /* Klasse, deren ProcessJob-Methode
    * den Delegate aufruft */
   public class WorkerClass
   {
      /* Deklaration der Ereignis-Eigenschaft */
      public event LogHandler OnLog;

      public void ProcessJobs()
      {
         // irgend etwas ausführen
```

```
            System.Threading.Thread.Sleep(1000);
            /* Loggen */
            if (OnLog != null)
            {
                /* Instanz der LogEventArgs-Klasse
                 * erzeugen  */
               LogEventArgs e =
                  new LogEventArgs("Done the first job");
               /* Ereignismethode aufrufen */
               OnLog(this, e);
            }
            // etwas anderes ausführen
            System.Threading.Thread.Sleep(1000);
            /* Loggen */
            if (OnLog != null)
            {
               LogEventArgs e =
                  new LogEventArgs("Done the second job");
               OnLog(this, e);
            }
            // etc.
         }
      }
}
```

Die benutzende Anwendung muss nun ein Behandlungsmethode erzeugen (wie schon bei Delegates) und diese übergeben:

```
using System;
using Library;

namespace App
{
   class Start
   {
      /* Methode für das Ereignis */
      static void LogHandler(object sender,
         LogEventArgs e)
      {
```

```
            Console.WriteLine(e.Message);
        }

        [STAThread]
        static void Main(string[] args)
        {
            /* Instanz der Klasse WorkerClass erzeugen */
            WorkerClass worker = new WorkerClass();
            /* Instanz des einen Log-Handlers übergeben */
            worker.OnLog += new LogHandler(LogHandler);
            /* Die Methode aufrufen, die ihre Arbeit
             * protokollieren soll.*/
            worker.ProcessJobs();

            Console.WriteLine("Ready");
        }
    }
}
```

5 Grundlegende Programmiertechniken

Eine der wichtigsten grundlegenden Aufgaben beim Programmieren ist die Bearbeitung von Zeichenketten. Aber auch die Arbeit mit Datumswerten und die Berücksichtigung kultureller Unterschiede gehören zu den grundlegenden Programmiertechniken, die in diesem Kapitel beschreiben werden.

5.1 Zeichenketten bearbeiten

Zeichenketten können Sie sehr gut über die Methoden der `string`-Klasse bearbeiten. Tabelle 5.2 listet die wichtigsten Instanzmethoden, Tabelle 5.1 die einzige wichtige Eigenschaft auf. Sie können Strings aber auch mit den Methoden der `Regex`-Klasse über reguläre Ausdrücke sehr flexibel durchsuchen und bearbeiten. Die Grundlagen davon beschreibe ich ab Seite 243.

Beachten Sie, dass die Instanzmethoden zur Bearbeitung des Strings den gespeicherten String nicht verändern, sondern immer einen neuen, bearbeiteten String zurückgeben.

Wenn Sie den gespeicherten String verändern wollen, müssen Sie den Rückgabewert der Methode auf diesen zuweisen:

```
string s = "   abc   ";
s = s.Trim();
```

Ab Seite 238 finden Sie Beispiele zum Umgang mit den String-Methoden.

Eigenschaft	Beschreibung
`Length`	liefert die Länge des Strings

Tabelle 5.1: Die einsame Instanzeigenschaft der string-Klasse

Methode	Beschreibung
`bool EndsWith(` `  string value)`	ermittelt, ob der String mit dem übergebenen String endet.

`int IndexOf(`
`  {string | char} value`
`  [, int startIndex]`
`  [, int count])`
	Mit dieser Methode können Sie den Index eines Zeichens oder Strings ermitteln. Wenn Sie *startIndex* angeben, wird ab diesem Index gesucht, ansonsten ab dem ersten Zeichen. Die Indizierung beginnt, wie in C# üblich, bei 0. Geben Sie *count* an, werden nur die Anzahl Zeichen durchsucht, die Sie hier angegeben haben. Wird das Zeichen bzw. der String nicht gefunden, resultiert der Wert -1.

`int IndexOfAny(`
`  char[] anyOf`
`  [, int startIndex]`
`  [, int count])`
	Über diese Methode können Sie den Index eines von mehreren Zeichen ermitteln, die Sie im ersten Argument als `char`-Array übergeben.

`string Insert(`
`  int startIndex,`
`  string value)`
	fügt einen String an der angegebenen Position ein und gibt diesen String zurück.

`int LastIndexOf(`
`  {string | char} value`
`  [, int startIndex]`
`  [, int count])`
	ermittelt ähnlich `IndexOf` den Index eines Zeichens oder eines Strings. `LastIndexOf` ermittelt aber den Index des letzten Vorkommens.

`int LastIndexOfAny(`
`  char[] anyOf`
`  [, int startIndex]`
`  [, int count])`
	ermittelt ähnlich `IndexOfAny` den Index eines von mehreren Zeichen. `LastIndexOfAny` ermittelt aber den Index des letzten Vorkommens.

`string PadLeft(`
`  int totalWidth`
`  [, char paddingChar])`
	füllt einen String nach links mit Leerzeichen oder dem in `paddingChar` angegebenen Zeichen auf, bis die in *totalWidth* angegebene Länge erreicht ist und gibt diesen String zurück.

`string PadRight(`
`  int totalWidth`
`  [, char paddingChar])`
	füllt einen String nach rechts auf.

Tabelle 5.2: Die wichtigsten Instanzmethoden der string-Klasse

Methode	Beschreibung

`string Remove(`
` int startIndex,`
` int count)`
	gibt den um einen Teilstring reduzierten String zurück.

`string Replace(`
`  char oldChar,`
`  char newChar)`
`string Replace(`
`  string oldValue,`
`  string newValue)`
	ersetzt alle Vorkommen eines Zeichens oder Strings in einem String durch ein anderes Zeichen bzw. einen anderen String und gibt diesen String zurück.

`string[] Split(`
`  char[] separator,`
`  int count)`
`string[] Split(`
`  params char[] separator)`
	splittet einen String an den im ersten Argument übergebenen Trennzeichen auf. `Split` gibt ein Array aus Strings mit den ermittelten Teilstrings zurück. So können Sie einen String sehr einfach in einzelne Teilstrings zerlegen.
	Im ersten Argument übergeben Sie ein Array aus `char`-Werten mit den möglichen Trennzeichen. Alternativ können Sie hier auch ein leeres Array oder `null` übergeben. C# verwendet dann ein Komma und verschiedene Escape-Sequenzen (\n, \r, \t) als Trennzeichen. Im zweiten Argument können Sie festlegen, wie viele Teilstrings maximal zurückgegeben werden.

`bool StartsWith(` ` string value)`	ermittelt, ob der String mit einem bestimmten String beginnt.

`string Substring(`
` int startIndex,`
` [int length]);`
	ermittelt einen Teilstring. Wenn Sie *length* angeben, wird der Teilstring von *startIndex* an mit der angegebenen Länge zurückgegeben, ansonsten bis zum Ende des String.

`char[] ToCharArray(`
` [int startIndex,]`
` [int length]);`
	gibt den (Teil-)String als `char`-Array zurück.

Tabelle 5.2: Die wichtigsten Instanzmethoden der string-Klasse

Methode	Beschreibung
`string ToLower( [CultureInfo culture])` `string ToUpper( [CultureInfo culture])`	gibt den String in Klein- bzw. Großschreibung zurück. Sie können ein Objekt der Klasse `System.Globalization.CultureInfo` übergeben, um die bei der Konvertierung zu verwendende Kultur zu definieren (in verschiedenen Kulturen werden Zeichen unterschiedlich konvertiert). Übergeben Sie kein Objekt, verwendet der Compiler die aktuelle Kultur.

`string Trim( [params char[] trimChars])`
`string TrimEnd( [params char[] trimChars])`
`string TrimStart( [params char[] trimChars])`
	über diese Methode können Sie bestimmte Zeichen aus dem String entfernen, die rechts und/oder links stehen. Übergeben Sie keine Zeichen, werden alle Whitespace-Zeichen (das sind Zeichen, die im Ausdruck nicht sichtbar sind) entfernt. Zu diesen Zeichen gehört z. B. das Leerzeichen, aber auch das Carriage Return. Alternativ können Sie im ersten Argument ein Array mit Zeichen übergeben, die entfernt werden sollen.

Tabelle 5.2: Die wichtigsten Instanzmethoden der string-Klasse

Die `string`-Klasse besitzt daneben noch einige Klassenmethoden. Die wichtigsten finden Sie in Tabelle 5.3. Beachten Sie, dass ich Beispiele zu diesen Methoden ab Seite 238 beschreibe.

Methode	Beschreibung
`int Compare( string strA, string strB [, bool ignoreCase], [, CultureInfo culture])`	vergleicht einen String mit einem anderen. Per Voreinstellung wird dabei Groß- und Kleinschreibung berücksichtigt (binärer Vergleich). Wenn Sie im dritten Argument `true` angeben, erfolgt der Vergleich ohne Berücksichtigung der Groß- und Kleinschreibung. In einer anderen Variante können Sie neben den zu vergleichenden Strings noch einen Index angeben, ab dem verglichen werden soll.

Methode	Beschreibung

`int Compare(`
`  string strA,`
`  int indexA,`
`  string strB,`
`  int indexB,`
`  [, int length]`
`  [, bool ignoreCase],`
`  [, CultureInfo culture])`
	Die Rückgabe ist wie bei Vergleichsmethoden üblich. Ein Wert kleiner als o bedeutet, dass String *A* kleiner als String *B*. Bei o sind beide Strings sind gleich groß. Ein Wert größer als o bedeutet, dass String *A* größer ist als String *B*. Beim Vergleich werden Kulturinformationen berücksichtigt. Wenn Sie nicht die Systeminformationen verwenden wollen, können Sie diese im letzten Argument separat angeben.

`int CompareOrdinal(`
`  string strA,`
`  string strB)`
 	
`int CompareOrdinal(`
`  string strA,`
`  int indexA,`
`  string strB,`
`  int indexB,`
`  int length)`
	vergleicht ähnlich `Compare`, aber ohne Berücksichtigung der Kultur und mit Berücksichtigung der Groß-/Kleinschreibung (rein binärer Vergleich).

`string Copy(` `  string str)`	gibt eine neue Instanz eines Strings mit der Kopie des übergebenen Strings zurück. Beachten Sie, dass eine einfache Zuweisung wie `s1 = s2` zunächst keine neue Instanz erzeugt, sondern beide Referenzen auf dieselbe Instanz zeigen lässt (erst bei Veränderung eines der beiden Strings wird eine separate Instanz erzeugt) . `Copy` erzeugt dagegen immer eine neue Instanz.

Methode	Beschreibung

`string Format(`
` string format`
` object arg0`
` [, object arg1]`
` [, object arg2])`
`string Format(`
` string format`
` params object[] args)`
`string Format(`
` IFormatProvider`
` provider,`
` string format`
` params object[] args)`
	formatiert den in *format* angegebenen String unter Berücksichtigung der Werte, die in *args0*, *args1* und *args2* bzw. in *args* (als Array) angegeben sind. Alternativ können Sie am ersten Argument einen Format-Provider übergeben. Das Formatieren wird ab Seite behandelt.

`string Join(`
` string separator,`
` string[] value`
` [, int startIndex,`
` int count])`
	`Join` erzeugt aus einem String-Array einen einzelnen String. Die im Array gespeicherten Strings werden durch den angegebenen Separator getrennt. Über *startIndex* können Sie einen Index angeben, ab dem das Array berücksichtigt werden soll. *count* gibt dann die Anzahl der Elemente an, die im String erscheinen sollen.

Tabelle 5.3: Die wichtigsten Klassenmethoden der string-Klasse

5.1.1 Teilstrings extrahieren und Strings kürzen

Mit den Methoden `Trim`, `TrimStart` und `TrimEnd` können Sie links, rechts und beidseitig Zeichen aus einem String entfernen. Wenn Sie diesen Methoden keine besonderen Zeichen übergeben, werden Whitespaces (im Druck nicht sichtbare Zeichen) entfernt (Leerzeichen, Tabulatoren, Carriage Return etc.):

```
string s = "  \t\r\n a b c  ";
Console.WriteLine(s.Trim()); // "a b c"
Console.WriteLine(s.TrimStart()); // "a b c  "
Console.WriteLine(s.TrimEnd()); // "  \t\r\n a b c"
```

Sie können im ersten Argument aber auch spezifizieren, welche Zeichen entfernt werden sollen:

```
s = "--**a-b-c**--";
Console.WriteLine(s.Trim('-', '*')); // "a-b-c"
```

Über die `Substring`-Methode können Sie Teilstrings extrahieren:

```
Console.WriteLine(s.Substring(3)); // "def"
Console.WriteLine(s.Substring(0, 3)); // "abc"
Console.WriteLine(s.Substring(2, 2)); // "cd"
```

5.1.2 Strings vergleichen und Teilstrings suchen

Wenn Sie Strings über den Vergleichsoperator miteinander vergleichen, unterscheidet C# Groß- und Kleinschreibung:

```
string s1 = "abc", s2 = "ABC";
if (s1 == s2) // wird niemals wahr
   ...
```

Um die Groß- und Kleinschreibung bei Stringvergleichen zu vernachlässigen, müssen Sie die statische `Compare`-Methode verwenden und im dritten Argument `true` angeben:

```
string s1 = "abc", s2 = "ABC";
if (String.Compare(s1,s2,true) == 0) // wird jetzt wahr
   ...
```

Über diese Methode können Sie auch Teile von Strings vergleichen:

```
string s1 = "abcXYdef", s2 = "123xy456";
if (String.Compare(s1,3,s2,3,2,true) == 0) // ist wahr
  ...
```

Verschiedene Methoden erlauben das Suchen von Teilstrings in einem String:

```
/* Ermitteln, ob ein String mit einer bestimmten
 * Zeichenkette beginnt */
string s = "abcxyz";
if (s.StartsWith("abc")) // true
   ...
/* Ermitteln, ob ein String mit einer bestimmten
```

```
 * Zeichenkette endet */
s = "abcxyz";
if (s.EndsWith("xyz")) // true
   ...
/* Ermitteln der Position einer bestimmten
 * Zeichenkette im String */
s = "0aa3aa6aa9";
// das erste Vorkommen suchen
int i = s.IndexOf("aa"); // ergibt 1
s = "0aa3aa6aa9";
// ab dem dritten Zeichen suchen
i = s.IndexOf("aa",2); // ergibt 4
s = "0aa3aa6aa9";
// das letzte Vorkommen suchen
i = s.LastIndexOf("aa"); // ergibt 7

/* Ermitteln der Position eines von mehreren
 * Zeichen im String */
// das erste Vorkommen suchen
string firstName = "Fred-Bogus";
i = firstName.IndexOfAny(
   new Char[] {'-', ' ', '_'}); // ergibt 4

// das letzte Vorkommen suchen
string fullName = "Fred-Bogus Trumper";
i = fullName.LastIndexOfAny(
   new Char[] {'-', ' ', '_'}); // ergibt 10
```


5.1.3 Strings manipulieren

Teilstrings ersetzen

Wenn Sie Teilstrings nicht »von Hand« (über die anderen Methoden der `string`-Klasse) ersetzen wollen (was sehr aufwändig ist), können Sie dazu einfach die `Replace`-Methode verwenden. `Replace` ist erstaunlich einfach. Sie können damit *alle* Vorkommen eines Strings oder eines Zeichens durch einen anderen String bzw. ein anderes Zeichen ersetzen:

```
string s = "0aa3aa6aa9";
Console.WriteLine(s.Replace("aa", "xx"));
// ergibt "0xx3xx6xx9"
```

Damit können Sie auch Zeichen aus dem String löschen:

```
// Zeichen löschen (durch einen Leerstring ersetzen)
string s = "a-b-c-d-e-f";
Console.WriteLine(s.Replace("-", ""));
// ergibt "abcedf"
```

Im Prinzip ist das auch das, was in der Praxis benötigt wird. Selten müssen Sie in einem String auch Ersetzungen nur ab einer bestimmten Position oder mit einer bestimmten Maximalanzahl ausführen. Die `Replace`-Methode der `string`-Klasse erlaubt dies allerdings nicht. Dazu müssen Sie eine Instanz der `StringBuilder`-Klasse verwenden, die ich ab Seite 243 beschreibe.

Strings nach links oder rechts auffüllen

Mit den Methoden `PadLeft` und `PadRight` können Sie einen String links oder rechts mit einer bestimmten Anzahl von Zeichen auffüllen:

```
string s = "abc";
Console.WriteLine(s.PadLeft(5)); // ergibt "  abc"
Console.WriteLine(s.PadRight(5)); // ergibt "abc   "
Console.WriteLine(s.PadLeft(2)); // ergibt "abc"
Console.WriteLine(s.PadRight(5, '-'));
 // ergibt "abc---"
```

Strings einfügen und löschen

Über die `Insert`-Methode können Sie einen anderen String einfügen:

```
s = "ab";
Console.WriteLine(s.Insert(1, "XY")); // ergibt "aXYb"
```

Über `Remove` löschen Sie einen Teilstring:

```
s = "aXYb";
Console.WriteLine(s.Remove(1,2)); // ergibt "ab"
```

Strings in Groß- und Kleinschreibung umwandeln

Die Methoden `ToUpper` und `ToLower` wandeln einen String in Groß- bzw. Kleinschreibung um. Diese Methoden berücksichtigen dabei länderspezifische Eigenheiten. Wenn Sie nichts weiter angeben, werden die Systemeinstellungen verwendet:

```
s = "abc-ABC-äöü-ÄÖÜ";
Console.WriteLine(s.ToUpper());
// ergibt "ABC-ABC-ÄÖÜ-ÄÖÜ"
Console.WriteLine(s.ToLower());
// ergibt "abc-abc-äöü-äöü"
```

Sie können im ersten Argument aber auch ein `CultureInfo`-Objekt übergeben, um eine bestimmte Kultur zu berücksichtigen. Wie Sie damit umgehen, zeige ich ab Seite 253.

5.1.4 Trennen und Zusammensetzen von Strings

In der Praxis müssen Sie Strings häufig in einzelne Teilstrings zerlegen. Wenn die Teilstrings durch bestimmte Trennzeichen voneinander getrennt sind, können Sie dazu einfach die `Split`-Methode verwenden. `Split` trennt einen String in mehrere einzelne Strings auf, die dann in einem Array zurückgegeben werden.

Im ersten (`params`-)Argument können Sie einfach die einzelnen Zeichen übergeben, die Trennzeichen darstellen. Die Rückgabe der Methode weisen Sie einem String-Array zu:

```
string gasPersons[1] = "Lexa,Philo;Morris,Joan;Harry";
string[] gasPersonArray = gasPersons.Split(',', ';');
foreach (string name in gasPersonArray)
   Console.WriteLine(name);
```

Die Ausgabe des Programms ist (wie erwartet):

```
Lexa
Philo
Morris
Joan
Harry
```

1. Personen aus: Matt Ruff: G.A.S. Die Trilogie der Stadtwerke.

Alternativ können Sie im zweiten Argument auch eine Maximalanzahl für die Trennungen angeben (wenn Sie im String nur eine bestimmte Anzahl Teilstrings erwarten). Dann müssen Sie im ersten Argument allerdings ein `char`-Array übergeben (weil `params`-Argumente nur an letzter Stelle erlaubt sind):

```
gasPersonArray = gasPersons.Split(new char[] {';'},2);
foreach (string name in gasPersonArray)
   Console.WriteLine(name);
```

Die Ausgabe ist nun wie folgt:

```
Lexa, Philo
Morris,Joan;Harry
```

Wenn Sie einen mit `Split` zerlegten String (z. B. nach einer Bearbeitung) wieder zusammenfügen wollen, können Sie die `Join`-Methode verwenden. `Join` ist allerdings eine statische Methode. Um die Arbeitsweise von `Join` zu demonstrieren, verwendet das folgende Beispiel ein selbst definiertes String-Array:

```
string[] foolPersonArray[2] = new String[]
   {"Stephen", "Aurora", "Kalliope"};
string foolPersons = string.Join(",", foolPersonArray);
Console.WriteLine(foolPersons);
```

Dieses Beispiel ergibt den String "Stephen,Aurora,Kalliope".

Sie können das Array auch ab einem bestimmten Index mit einer bestimmten Anzahl Elementen in einen String umwandeln:

```
foolPersons = string.Join(",", foolPersonArray,1,2);
// ergibt "Aurora,Kalliope"
```

5.1.5 Reguläre Ausdrücke

Reguläre Ausdrücke (Regular Expressions) sind eine sehr mächtige Technik, über die Sie in Zeichenketten sehr flexibel suchen und ersetzen (!) können. Das .NET-Framework stellt dazu im Namensraum `System.Text.RegularExpressions` einige Klassen zur Verfügung.

2. Personen aus: Matt Ruff: Fool on the hill.

Sie werden sicherlich schon öfter unter Windows Wildcards verwendet haben, um bestimmte Dateien zu finden (z. B. *.doc für alle Word-Dokumente). Dann haben Sie schon einen einfachen regulären Ausdruck eingesetzt. Mit regulären Ausdrücken können Sie aber nach viel komplexeren Bedingungen suchen als mit Wildcards. So können Sie z. B. eine Zeichenkette daraufhin überprüfen, ob diese eine gültige E-Mail-Adresse enthält.

Reguläre Ausdrücke sind wesentlich mächtiger, als ich hier darstellen kann. immerhin gibt es ganze Bücher darüber. Spezielle Features, wie das Laden einer Fundstelle in einen Puffer und deren spätere Verwendung im String (Backreferences), kann ich hier nicht beschreiben. Vergleichen Sie die Informationen, die Sie in der .NET-Framework-Hilfe unter REFERENCE / REGULAR EXPRESSION LANGUAGE ELEMENTS. *Im Internet finden Sie an der Adresse etext.lib.virginia.edu/helpsheets/regex.html eine recht gute Beschreibung.*

Funktionsweise von regulären Ausdrücken

Bei regulären Ausdrücken vergleichen Sie ein Muster mit einem String. Das Muster "\d{5}" überprüft einen String z. B. daraufhin, ob dieser fünf Dezimalziffern hintereinander speichert. Grundsätzlich wird der String dabei von links nach rechts mit dem Muster verglichen. Dabei werden alle Fundstellen berücksichtigt. Der String "12345 xxy 67890" liefert beispielsweise zwei Fundstellen: "12345" und "67890". Sie können aber auch mit dem Musterzeichen ^ festlegen, dass der String mit dem Muster beginnen und mit $, dass der String mit dem Muster enden muss. Das Muster "^\d{5}$" führt demnach dazu, dass nur die Auswertung von Strings, die fünf Ziffern und sonst nichts beinhalten, erfolgreich ist.

Auswerten von regulären Ausdrücken mit C#

In C# können Sie einfach die statische `IsMatch`-Methode der `Regex`-Klasse verwenden, um zu überprüfen, ob ein String einem Muster entspricht:

```
using System.Text.RegularExpressions;
```

```
...

Console.Write("Geben Sie eine Postleitzahl ein: ");
string input = Console.ReadLine();
string pattern = @"^\d{5}$";
if (Regex.IsMatch(input, pattern))
   // Adresse ist OK
```

Der String ist im Beispiel als wortwörtliches Stringliteral angegeben, weil der Backslash nicht als Escape-Zeichen-Einleiter ausgewertet werden soll.

Beachten Sie, dass `IsMatch` *(entsprechend der Logik von regulären Ausdrücken) auch bei mehreren Fundstellen* `true` *ergibt. Wenn Sie die Musterzeichen ^ und/oder $ in dem Quellcode oben beispielsweise weglassen, würden auch Eingaben wie "Ka-Li 47475" oder "47475 Ka-Li" zum Erfolg führen.*

Wenn Sie die mehrere Fundstellen erwarten und diese extrahieren wollen, benötigen Sie eine Instanz der Klasse `Regex`. Im Konstruktor übergeben Sie das Muster:

```
using System.Text.RegularExpressions;

...
string pattern = @"\d{5}"; //
Regex regex = new Regex(pattern);
```

Um einen String damit zu vergleichen, verwenden Sie die `Match`-Methode, die eine Instanz der Klasse `Match` zurückgibt:

```
Match match = regex.Match(input);
if (match.Success)
   Console.WriteLine("Die PLZ ist in Ordnung.");
else
   Console.WriteLine("Das ist keine PLZ.");
```

Um mehrere Teilstellen zu extrahieren, verwenden Sie die `Matches`-Methode, die ein Objekt der Klasse `MatchCollection` zurückgibt. Über diese Auflistung können Sie die einzelnen Fundstellen durchgehen:

```
Console.WriteLine("Geben Sie einen String mit ");
Console.Write("mehreren dreistelligen Zahlen ein: ");
string input = Console.ReadLine();
string pattern = @"\d{3}";
regex = new Regex(pattern);
MatchCollection matches = regex.Matches(input);
foreach (Match m in matches)
{
   Console.WriteLine("Gefunden an Index " + m.Index +
        ": " + m.Value);
}
```

Ersetzen von Teilstrings

Über die `Replace`-Methode können Sie alle oder nur bestimmte Vorkommen eines Musters im String ersetzen. Die `Replace`-Methode der `Regex`-Klasse (die Sie statisch und als Instanzmethode verwenden können) besitzt einige Überladungen, von denen ich nur einige zeige:

```
/* x gefolgt von einer Ziffer durch ** ersetzen */
Regex regex = new Regex(@"x\d");
string input = "x1 x2 x3";
/* Alle Vorkommen ersetzen */
Console.WriteLine(regex.Replace(input, "*"));
// ergibt "* * *"
/* Nur die ersten zwei Vorkommen ersetzen */
Console.WriteLine(regex.Replace(input, "*", 2));
// ergibt "* * x3"
/* Zwei Vorkommen ab dem zweiten ersetzen */
Console.WriteLine(regex.Replace(input, "*", 2, 2));
// ergibt "x1 * *"
```

String splitten

Über die `Split`-Methode können Sie einen String an einem Muster auftrennen. `Split` gibt, wie die gleichnamige Methode der `string`-Klasse ein `string`-Array zurück:

```
Regex regex = new Regex(@"x\d");
string input = "AAx1BBx2CC";
```

```
/* An allen Fundstellen splitten */
string[] inputs = regex.Split(input);

foreach (string s in inputs)
   Console.WriteLine(s);

/* An maximal zwei Fundstellen splitten */
inputs = regex.Split(input, 2);

...

/* Ab Zeichen 3 maximal zwei Fundstellen splitten */
inputs = regex.Split(input, 2, 3);
```

Zeichen im Muster

Reguläre Ausdrücke können Escape-Zeichen enthalten ('\n', '\t' etc.) und spezielle Musterzeichen. Tabelle 5.4 listet die Musterzeichen auf.

Musterzeichen	Beschreibung
\	vor einem anderen Musterzeichen: Aufhebung der Sonderbedeutung des Musterzeichens.
.	genau ein beliebiges Zeichen außer \n.
*	kein, ein oder beliebig viele Zeichen.
?	kein oder ein Zeichen.
+	ein oder beliebig viele Zeichen.
[*Zeichenliste*]	genau ein Zeichen, das in der kommabegrenzten Liste angegeben ist. Sie können einzelne Zeichen angeben: [a,b,c] oder Zeichenbereiche [a-c]. Sie können beides kombinieren: [a-c,1,3,5,7]. Wenn Sie der Zeichenliste ein ^ voranstellen, bedeutet dies, dass die definierten Zeichen nicht vorkommen dürfen: [^a,b,c].
^	Am Beginn eines Musters definiert dieses Zeichen, dass das Muster am Anfang der Zeichenkette oder am Anfang einer Zeile stehen muss.

Musterzeichen	Beschreibung
$	Am Ende eines Musters definiert dieses Zeichen, dass das Muster am Ende der Zeichenkette oder am Ende einer Zeile stehen muss.
\A	Am Beginn eines Musters definiert dieses Zeichen, dass das Muster am Anfang der Zeichenkette Zeile stehen muss.
\z	Am Ende eines Musters definiert dieses Zeichen, dass das Muster am Ende der Zeichenkette stehen muss.
|	Alternative (logisches Oder).
{*n*}	genau *n* Zeichen, die links von {} spezifiziert sind. "\d{5}" steht z. B. für "genau 5 Dezimalziffern.
{*min*,}	mindestens *min* Zeichen, die links von {} spezifiziert sind.
{*min*,*max*}	mindestens *min* und maximal *max* Zeichen, die links von {} spezifiziert sind.
\d	eine Dezimalziffer.
\D	ein Zeichen, das keine Dezimalziffer ist.
\w	ein Wort-Zeichen im Bereich a-z, A-Z, 1-0.
\W	ein Nicht-Wort-Zeichen.
\s	ein Whitespace.
\s	ein Nicht-Whitespace.

Tabelle 5.4: Die wichtigsten Musterzeichen der regulären Ausdrücke

Einige Beispiele

Tabelle 5.5 zeigt einige Beispiele für reguläre Ausdrücke.

Muster	Bedeutung
^[\w\.-]{1,}@[\w\.-]{2,}\.\w{2,3}$	E-Mail-Adresse
^d{5}-d{4}$	US-Postleitzahl
^d{5}$	deutsche Postleitzahl

Tabelle 5.5: Einige Beispiele für reguläre Ausdrücke

5.1.6 Die StringBuilder-Klasse

Wenn Sie einen normalen String häufig manipulieren (kürzen, hinzufügen, ersetzen etc.), kostet dies viel Rechenzeit. Das liegt daran, dass Strings »unabänderlich« (immutable) sind. Eigentlich ist dieser Begriff etwas verwirrend, weil Sie Strings ja ohne Probleme verändern können:

```
string s = "Hi";
s += " Zaphod";
```

Der Compiler kann den String aber nicht an seinem originalen Speicherbereich verändern, weil dieser festgelegt ist. Er reserviert dann einfach einen neuen Speicherbereich und kopiert den geänderten String dort hinein.

Wenn Sie nun viele Manipulationen auf einem String ausführen, ist die Ausführungsgeschwindigkeit wegen der Reservier- und Kopiervorgänge recht mäßig.

Die Lösung für dieses Problem ist die `StringBuilder`-Klasse, die im Namensraum `System.Text` verwaltet wird. Die Verwendung dieser Klasse lohnt sich ab etwa zehn Stringmanipulationen. Unterhalb dieser Zahl ist der Overhead, den diese Klasse verursacht, größer als der Gewinn. Wenn Sie aber sehr viele Manipulationen ausführen, ist der Gewinn enorm. In einem Test, bei dem ich 50.000-mal ein Zeichen zu einem String hinzugefügt hatte, benötigte das Programm mit einer Instanz der `StringBuilder`-Klasse (inkl. Instanzierung) 1 bis 10 Millisekunden, mit einem echten String hingegen ca. neun Sekunden. Das ist ein Gewinn-Faktor von 1000!

Eine Instanz dieser Klasse speichert einen String und erlaubt verschiedene Operationen darauf (über Methoden). Die `StringBuilder`-Klasse reserviert allerdings mehr Speicher als eigentlich benötigt ist und ist deswegen in der Lage, den String direkt (und damit wesentlich schneller) zu manipulieren. Nach den erfolgten Manipulationen können Sie den fertigen String über die `ToString`-Methode einem einfachen `string`-Typ zuweisen.

Die `StringBuilder`*-Klasse ist im Wesentlichen für die Manipulation von Strings gedacht. Methoden, wie z. B.* `IndexOf`*, die lediglich Informationen liefern, werden Sie (auch in neueren Versionen des .NET-Framework) vergeblich suchen. Sofern Sie diese überhaupt suchen ...*

Instanzen erzeugen

Bei der Erzeugung einer Instanz der `StringBuilder`-Klasse können Sie verschiedene Initialisierungen vornehmen:

```
System.Text.StringBuilder sb;

/* Instanz mit einem geschätzten Anfangs Platz von 255
 * Zeichen ohne Maximalangabe erzeugen */
sb = new System.Text.StringBuilder(255);

/* Instanz mit einem geschätzten Anfangs Platz von 255
 * Zeichen und einem maximalen Platz von 1024 Zeichen
 * erzeugen */
sb = new System.Text.StringBuilder(255, 1024);

/* Instanz erzeugen und mit einem String
 * initialisieren */
sb = new System.Text.StringBuilder(
   "Das ist ein String");

/* Instanz erzeugen und mit einem String
 * initialisieren; Maximalgröße festlegen */
sb = new System.Text.StringBuilder(
   "Das ist ein String", 1024);

/* Instanz erzeugen und mit einem Teilstring
 * initialisieren; Maximalgröße festlegen */
sb = new System.Text.StringBuilder(
  "Das ist ein String",12,6,1024);
```

Wenn Sie die Maximalgröße festlegen, darf der gespeicherte String nicht größer sein. Würde der String über eine Operation größer werden, würde die CLR eine Ausnahme vom Typ `ArgumentOutOfRangeExcep-`

tion erzeugen. Geben Sie die Maximalgröße nicht an, können Sie den String fast beliebig vergrößern. Die Größe ist in der zur Zeit der Drucklegung dieses Buchs aktuellen Version des .NET-Framework auf dem Maximalwert von int beschränkt (2.147.483.647), vorausgesetzt, der freie Arbeitsspeicher ist groß genug. Sie können eine angegebene Maximalgröße jederzeit über die EnsureCapacity-Methode verändern. In meinen Performancetests machte es nichts aus, ob ich die Anfangsgröße klein oder groß einstellte und ob ich eine Maximalgröße angab.

Strings an- und einfügen

Über die Append-Methode können Sie nun verschiedene Datentypen anfügen:

```
/* String anfügen */
sb.Append("Test");

/* Teilstring anfügen */
string s = "abcXYdef";
sb.Append(s,3,2); // nur "XY"

/* double-Wert anfügen */
double d = 1.234; // alle anderen Standardtypen sind
                  // ebenfalls möglich
sb.Append(d);

/* char-Array anfügen */
char[] charArray = new Char[] {'a', 'b', 'c'};
sb.Append(charArray);

/* Zeichen x-mal anfügen */
sb.Append('*', 10);
```

Über die AppendFormat-Methode können Sie einen String formatiert einfügen:

```
sb.AppendFormat("Heute ist '{0}', der {1}",
   DateTime.Today.DayOfWeek.ToString(),
   DateTime.Today.ToShortDateString());
```

Formatierungen werden ab Seite 255 behandelt.

Die `Insert`-Methode erlaubt das Einfügen der Standarddatentypen ähnlich der `Append`-Methode, nur dass Sie hier im ersten Argument zusätzlich den Startindex angeben:

```
sb = new System.Text.StringBuilder("Test");
sb.Insert(0, "Das ist ein "); // Am Anfang einfügen
```

Länge und Kapazität ermitteln

Über die `Length`-Eigenschaft erhalten Sie die Länge des gespeicherten Strings. Die Eigenschaft `MaxCapacity` gibt die maximal speicherbare Kapazität zurück:

```
Console.WriteLine("Länge: " + sb.Length +
   ", maximale Kapazität: " + sb.MaxCapacity);
```

Teilstrings löschen

Über die `Remove`-Methode können Sie Teilstrings löschen:

```
sb = new  System.Text.StringBuilder(
   "Das ist ein Test ist das");

/* die ersten zwölf Zeichen löschen */
sb.Remove(0, 12); // "Test ist das"

/* Ab dem fünften Zeichen bis zum Ende löschen */
sb.Remove(4, sb.Length-4); // "Test"
```

Teilstrings ersetzen

Ähnlich der gleichnamigen Methode der `string`-Klasse ermöglicht die `Replace`-Methode der `StringBuilder`-Klasse das Ersetzen eines Teilstrings durch einen anderen. Im ersten Argument geben Sie einen String oder ein Zeichen an, das ersetzt werden soll, im zweiten Argument einen String oder ein Zeichen, das den neuen Wert darstellt:

```
sb = new  System.Text.StringBuilder("0ab3ab6ab9");
sb.Replace("ab","*"); // "0*3*6*9"
...
sb = new  System.Text.StringBuilder("0ab3ab6ab9");
sb.Replace('a',"*"); // "0*b3*b6*b9"
```

Sie können im dritten (*startIndex*) und vierten Argument (*count*) auch einen Teilstring spezifizieren, in dem ersetzt werden soll:

```
sb = new  System.Text.StringBuilder("0ab3ab6ab9");
/* In Teilstring ab Position 3 mit Länge 3 ersetzen */
sb.Replace("ab","*",3,3); // "0ab3*b6ab9"
```

Den String ausgeben

Über die `ToString`-Methode geben Sie den String schließlich zurück:

```
string s = sb.ToString();
```

5.2 Berücksichtigung der Kultur

Viele Methoden, die mit Strings arbeiten, berücksichtigen kulturelle Informationen. Wenn Sie beispielsweise ein String-Array sortieren, wird dies in den USA wohl anders erfolgen als in Deutschland, weil einzelne Zeichen in den verschiedenen Ländern an anderer Stelle positioniert sind.

Diese Methoden ermöglichen die Spezifikation der Kultur. Wenn Sie diese nicht angeben, verwenden die Methoden (hoffentlich auch alle) die Systemeinstellungen. Wenn Sie Anwendungen schreiben, die in verschiedenen Ländern ausgeführt werden sollen, müssen Sie normalerweise nichts weiter machen.

Problematisch wird das Ganze aber in Webanwendungen, die auf einem Webserver ausgeführt wird. Diese Anwendungen würden dann die Ländereinstellung des Webservers verwenden und nicht die des Benutzers. Vielleicht wollen Sie dem Benutzer aber auch in normalen Anwendungen die Möglichkeit geben, die Kultur für Ihre Anwendungen zu spezifizieren.

Um die Kultur zu spezifizieren benötigen Sie ein Objekt der Klasse `System.Globalization.CultureInfo`. Diesem Objekt können Sie bei der Erzeugung eine ID übergeben, die die Kultur definiert. Diese ID kann ein String im standardisierten Format RFC 3066 (*www.rfc-editor.org*) sein:

```
System.Globalization.CultureInfo culture =
   new System.Globalization.CultureInfo("en-US");
```

Dieser String enthält normalerweise zwei bis drei klein geschriebene Zeichen, die die Sprache definieren und optional, getrennt durch einen Bindestrich, zwei bis drei Zeichen, die das Land spezifizieren. "de-DE" steht beispielsweise für »Deutsch – Deutschland« und "en-US" für »English – United States«. Alternativ können Sie eine der vordefinierten, ebenfalls standardisierten LCID-Konstanten (0x0007 für Deutschland und 0x0409 für USA) übergeben. Sie finden diese Werte in der Hilfe zur `CultureInfo`-Klasse.

Einige Kulturen sind grundsätzlich in drei Bereiche aufgeteilt: Eine »unveränderliche« (invariant) Kultur ist nicht mit einem spezifischen Land, aber mit der englischen Sprache verknüpft. Diese Kultur können Sie in Situationen einsetzen, in denen Sie keine spezielle Kultur ermitteln können, aber trotzdem eine einsetzen können. Erzeugen Sie eine unveränderliche Kultur, indem Sie einen Leerstring an den Konstruktor der `CultureInfo`-Klasse übergeben. Neutrale Kulturen sind mit einer Sprache, aber nicht mit einem Land verknüpft. "de" (Deutsch – Allgemein) ist z. B. eine solche Kultur, "de-DE" (Deutsch – Deutschland) ist dagegen eine spezifische Kultur. Spezifische Kulturen sind mit einer Sprache und einem Land verknüpft. Einige Methoden können nur mit spezifischen Kulturen verwendet werden.

Das `CultureInfo`-Objekt können Sie nun überall da übergeben, wo die Schnittstelle `IFormatProvider` erwartet wird (die von der `CultureInfo`-Klasse implementiert wird), z. B. der `Format`-Methode der String-Klasse:

```
double number = 1234.55; string s;
s = String.Format(culture,"{0:#,#0.00}",number));
// ergibt "1,234.55"
```

Besser ist aber, gleich die Kultur dem gesamten Thread zuzuweisen. Damit verwenden alle Methoden automatisch diese Kultur:

```
System.Threading.Thread.CurrentThread.CurrentCulture =
   culture;
s = DateTime.Now.ToShortDateString());
// ergibt z. B. 11/20/2001
```

5.3 Formatierungen

Viele Methoden, die mit Strings arbeiten oder Strings zurückgeben, können mit einem String verwendet werden, der Formatierspezifikationen enthält. Die `Format`-Methode der `string`-Klasse ist das beste Beispiel dafür. Der zu formatierende String kann gleich mehrere Spezifikationen enthalten:

```
int i = 10;
string s = String.Format(
   "Die Wurzel aus {0} ist gerundet {1:0.00}",
   i, Math.Sqrt(i));
```

Der String enthält hier zwei Formatspezifikationen, die in geschweiften Klammern angegeben sind. Bei solchen Methoden werden die Werte, die in diese Formatspezifikationen eingesetzt werden sollen, in einem `params`-Argument übergeben, wie es im Beispiel mit der Variablen *i* und dem Aufruf der Methode `Math.Sqrt` der Fall ist. Die Methode formatiert die Werte entsprechend und fügt diese an die Stelle der Formatspezifikation ein.

Die Syntax eines normalen Formatspezifizierers ist wie folgt:

{*Parameterindex*[, *Minimalbreite*][:*Formatstring*]}

Der Parameterindex bestimmt den Index des Arguments, das formatiert werden soll. Der Index beginnt wie immer bei o. In der optionalen Breite können Sie angeben, wie breit der erzeugte (Teil-)String minimal sein soll. Ist der hier angegebene Wert positiv, wird der String nach links mit Leerzeichen aufgefüllt, ansonsten nach rechts. Im Formatstring geben Sie schließlich die Formatierung an, falls Sie den Parameter formatiert ausgeben wollen.

Der Formatstring kann festgelegte Zeichen enthalten, die eine vordefinierte Formatierung spezifizieren, oder kann aus speziellen Formatierzeichen bestehen, die eine benutzerdefinierte Formatierung erlauben. Das Zeichen 'C' steht z. B. für eine Formatierung im länderspezifischen Währungsformat. Die Zeichenfolge "0.00" formatiert einen Wert mit führender o und zwei Nachkommastellen (wobei gleich noch gerundet wird). Die verwendbaren Zeichen sind sehr umfangreich. Sie finden diese Zeichen in der Hilfe, wenn Sie im Index nach »format specifiers« suchen.

Zahlformatierungen

Tabelle 5.6 zeigt die wichtigsten Zahlformatierungen. Beachten Sie, dass die Formatierungen länderspezifisch ausgeführt werden.

Format	Wert	Erläuterung	Ergebnis
C	100.955	Währung mit Standard-Dezimalstellen	"100,96 _"
C4	100.955	Währung	"100,9550 _"
F2	100.955	Zahl mit fester Dezimalstelle	"100,96"
F4	100.955	Zahl mit fester Dezimalstelle	"100,9550"
X	255	Hexadezimales Format	"FF"
0.00	0.345	Benutzerdefiniert	"0,35"
0.00	12.345	Benutzerdefiniert	"12,35"
0.00	1234.567	Benutzerdefiniert	"1234,57"
#,#0.00	1234.567	Benutzerdefiniert	"1.234,57"
#,#.00	0.567	Benutzerdefiniert	",57"

Tabelle 5.6: Die wichtigsten Zahlformatierungen

Ein paar Beispiele sollen die Arbeit mit Formatstrings verdeutlichen. Die Anweisungen nutzen die Formatierfähigkeit der `WriteLine`-Methode der `Console`-Klasse:

```
Console.WriteLine("{0:C}", 100.955);
// ergibt "100,96 _"
Console.WriteLine("{0:0.00}", 0.345);
// ergibt "0,35"
Console.WriteLine("{0:#,#0.00}", 1234.567);
// ergibt "1.234,57"
```

Datumsformatierungen

Tabelle 5.7 stellt die wichtigsten Datumsformatierungen dar. Die Formatierungen beziehen sich auf das Datum 31.12.2005 12:30. Beachten Sie, dass nur einige spezielle benutzerdefinierte Formate nicht länderspezifisch sind.

Format	Erläuterung	Ergebnis
D	langes Datumsformat	"Samstag, 31. Dezember 2005"
d	kurzes Datumsformat	"31.12.2005"
g	kurzes Datumsformat mit kurzer Zeit	"31.12.2005 12:30"
t	kurzes Zeitformat	"12:30"
T	langes Zeitformat	"12:30:00"
u	ISO-Format	"2005-12-31 12:30:00Z"
dd.MM.yy	benutzerspezifisch	"31.12.05"
dddd, dd. MMMM yyyy	benutzerspezifisch	"Samstag, 31. Dezember 2005"
yyyy-MM-dd	benutzerspezifisch	"2005-12-31"
hh:mm:ss	benutzerspezifisch	"12:30:00"

Tabelle 5.7: Die wichtigsten Datumsformate

5.4 Datumswerte bearbeiten

Datumswerte speichern Sie in einer Instanz der `DateTime`-Klasse. Über verschiedene Instanzmethoden der `DateTime`-Klasse können Sie mit den gespeicherten Datumswerten arbeiten. Die wichtigsten finden Sie in Tabelle 5.8.

Methode	Beschreibung

`DateTime AddDays(`
`  double` *`value`*`)`
`DateTime AddHours(`
`  double` *`value`*`)`
`DateTime AddMinutes(`
`  double` *`value`*`)`
`DateTime AddMonths(`
`  double` *`value`*`)`
`DateTime AddYears(`
`  double` *`value`*`)`
	gibt ein um die angegebene Anzahl Tage/Stunden/Minuten/Monate/Jahre addiertes Datum als `DateTime`-Objekt zurück.

Methode	Beschreibung

`string ToLongDateString()`
`string ToLongTimeString()`
`string ToShortDateString()`
`string ToShortTimeString()`
	Diese Methoden geben das Datum als kurzer oder langer Datums-/Zeitstring zurück.

Tabelle 5.8: Die wichtigsten Instanzmethoden der DateTime-Klasse

Einige Instanzeigenschaften liefern Informationen zum Datum.

Eigenschaft	Bedeutung
`Day, Hour, Minute, Month, Second, Year`	Über diese Eigenschaften können Sie die Teile des Datums ermitteln.
`DayOfWeek`	gibt den Wochentag als `DayOfWeek`-Typ zurück. `DayOfWeek ist eine Aufzählung mit den Werten Monday, Tuesday, Wednesday, Thursday, Friday, Saturday und Sunday.`
`DayOfYear`	gibt die Nummer des Tages bezogen auf das Jahr als `int`-Wert zurück

Tabelle 5.9: Die wichtigsten Instanzeigenschaften der DateTime-Klasse

Die `DateTime`-Klasse besitzt daneben noch einige Klasseneigenschaften und -Methoden, die in Tabelle 5.10 dargestellt werden.

Methode	Beschreibung
`int DaysInMonth( int` *`year`*`, int` *`month`*`)`	ermittelt die Anzahl der Tage eines Monats
`bool IsLeapYear( int` *`year`*`)`	ermittelt, ob ein Jahr ein Schaltjahr ist
`DateTime Parse( string` *`s`* `[...])`	Über diese Methode können Sie einen String in einen `DateTime`-Wert umwandeln. Neben dem String können Sie in den zusätzlichen Argumenten die zu verwendenden Kulturinformationen des Datums definieren. Das Datum darf in einer beliebigen, zur Kultur passenden Form übergeben werden.

Methode	Beschreibung
`DateTime ParseExact( string s, string format ...)` `DateTime ParseExact( string s, string[] formats ...)`	`ParseExact` wandelt einen String in einen `DateTime`-Wert um. Der String muss einem der im zweiten Argument übergebenen Formate entsprechen. Diese Methode erwartet in den weiteren Argumenten wieder Informationen zur verwendeten Kultur.

Tabelle 5.10: Die wichtigsten Klassenmethoden der DateTime-Klasse

Über einige Klasseneigenschaften können Sie Informationen abrufen (Tabelle 5.11).

Eigenschaft	Bedeutung
`Now`	gibt ein `DateTime`-Objekt mit dem aktuellen Datum inklusive Zeit zurück
`Today`	gibt ein `DateTime`-Objekt mit dem aktuellen Datum ohne Zeit zurück

Tabelle 5.11: Die wichtigsten Klasseneigenschaften der DateTime-Klasse

Die folgenden Beispiele sollen den Umgang mit der `DateTime`-Klasse verdeutlichen:

```
DateTime d1 = new DateTime(2005,12,31,12,30,0);
// 31.12.2005 12:30
DateTime d2;

/* 100 Tage aufaddieren */
d2 = d1.AddDays(100);

/* 3 Monate aufaddieren */
d2 = d1.AddMonths(3);
Console.WriteLine("+ 3 Monate: {0}", d2.ToString());

/* 3 Monate abziehen */
```

```
d2 = d1.AddMonths(-3);
Console.WriteLine("- 3 Monate: {0}", d2.ToString());

/* Wochentag ermitteln */
DayOfWeek dayOfWeek = d1.DayOfWeek;
Console.WriteLine("Tag in der Woche {0}: {1}",
   (int)dayOfWeek, dayOfWeek.ToString());

/* Tag im Jahr ermitteln */
Console.WriteLine("Tag im Jahr: {0}", d1.DayOfYear);

/* Tag im Monat ermitteln */
Console.WriteLine("Tag im Monat: {0}", d1.Day);

/* Monat ermitteln */
Console.WriteLine("Monat: {0}", d1.Month);

/* Jahr ermitteln */
Console.WriteLine("Jahr: {0}", d1.Year);
```


Datumswerte vergleichen

Die `DateTime`-Klasse überlädt die Vergleichoperatoren. Sie können Datumswerte also direkt vergleichen:

```
DateTime d = new DateTime(2005,12,31);
if (DateTime.Today >= d1)
   Console.WriteLine("Schön, dass Sie dieses " +
      "Beispiel noch Ende 2005 testen.");
else
   Console.WriteLine("Schade, da müssen wir wohl " +
      "noch warten ...");
```

Mit Zeitspannen arbeiten

Die Differenz zwischen zwei Datumswerten wird über den Typ `TimeSpan` (Zeitspanne) verwaltet. Über verschiedene Eigenschaften erhalten Sie Informationen, über verschiedene Methoden können Sie die Zeitspanne beeinflussen. Ich kann hier nicht alle Elemente dieser Klasse auflisten und zeige nur, wie Sie beispielsweise die Differenz

zwischen zwei Datumswerten berechnen. Die `DateTime`-Klasse überlädt den Subtraktionsoperator, der einen `TimeSpan`-Wert zurückgibt:

```
TimeSpan t;
DateTime d = new DateTime(2005,12,31);
t = d - DateTime.Today;
Console.WriteLine("Zwischen " +
   DateTime.Today.ToString() +
   " und " + d.ToString() + " liegen");
Console.WriteLine("{0} Tage,", t.TotalDays);
Console.WriteLine("{0} Stunden,", t.TotalHours);
Console.WriteLine("{0} Minuten,", t.TotalMinutes);
Console.WriteLine("{0} Sekunden.", t.TotalSeconds);
```

5.5 Mathematische Berechnungen

Über die statischen Methoden der `Math`-Klasse können Sie die verschiedensten mathematischen Berechnungen durchführen. Tabelle 5.12 listet einfach die wichtigsten auf. Ich denke, Sie können auch ohne Beispiele damit umgehen.

Methode	Beschreibung
`Abs`	Liefert den Absolutwert der übergebenen *Zahl* (= ohne Vorzeichen)
`Cos`, `Sin`, `Tan` `Acos`, `Asin`, `Atan`	Berechnen Kosinus, Sinus, Tangens, Arcuskosinus, Arcussinus und Arcustangens. Übergabe: Gradangabe in Bogenmaß (Bogenmaß = Grad * PI / 180).
`Exp`	Liefert den Exponenten aus übergebenen *Zahl* (Umkehrfunktion zu `Log`)
`Log`	Liefert den natürlichen Logarithmus der übergebenen *Zahl*
`Max`, `Min`	Liefern den kleineren bzw. größeren von zwei übergebenen Werten
`PI`	Diese Eigenschaft gibt den annähernd genauen Wert von PI zurück.

Methode	Beschreibung
`Pow`	liefert die Potenz einer Zahl. Im ersten Argument übergeben Sie die Basis, im zweiten den Exponenten. `Pow`(2,3) bedeutet so viel wie 2^3.
`Round`	`Round` rundet eine Zahl auf eine anzugebende Anzahl Dezimalstellen, allerdings leider nicht so, wie wir es gewohnt sind. Im ersten Argument geben Sie die zu rundende Zahl an. Das zweite Argument bestimmt, auf wie viele Stellen hinter dem Komma gerundet wird. Leider rundet `Round` mathematisch (und nicht kaufmännisch). Beim mathematischen Runden wird, wenn der Ziffer, auf die gerundet wird, genau eine 5 folgt, aufgerundet, wenn diese Ziffer ungerade ist, und abgerundet, wenn diese Ziffer gerade ist. So rundet `Round`(1.35, 1) auf 1,4 und `Round`(1.45, 1) ebenfalls auf 1,4. `Round`(1.45001, 1) rundet allerdings auf 1.5. So ist sichergestellt, dass beim Runden nicht ein Ungleichgewicht zu den aufgerundeten Zahlen hin entsteht (wie es beim kaufmännischen Runden der Fall ist).
`Sqrt`	Liefert die Quadratwurzel der übergebenen Zahl

Tabelle 5.12: Die wichtigsten Methoden und Eigenschaften der Math-Klasse

5.6 Massendaten in Auflistungen speichern

Häufig müssen Sie eine Vielzahl gleichartiger Daten speichern und mit diesen Daten arbeiten. Sinnvoll ist auf jeden Fall das Speichern in Objekten, deren Klassen Sie den Daten entsprechend deklarieren. Wenn Sie wissen, wie viele Objekte zu speichern sind, können Sie einfach mehrere Variablen oder ein Array verwenden. In der Praxis wissen Sie dies aber meist nicht, hier kommt es z. B. recht häufig vor, dass Sie die Daten aus einer Datenbanktabelle in den Arbeitsspeicher lesen, um später sequentiell oder (extrem schnell) über einen Schlüssel auf die gespeicherten Objekte zuzugreifen. Besonders bei 3-Schichten-Anwendungen, deren zweite Schicht (die »Geschäftslogik-Schicht«) die Logik zum Zugriff auf die Daten implementiert, sind solche Objektlisten sehr wichtig.

Wenn Sie also mehrere Objekte speichern müssen, verwenden Sie entweder ein Array oder eine der vielen Auflistungen (Collections) des .NET-Frameworks. Arrays bieten recht einfache Speichermöglichkeiten und besitzen eine bei der Deklaration festgelegte Größe, Auflistungen bringen erweiterte Möglichkeiten mit, wie z. B. das dynamische Erweitern und die Assoziation eines gespeicherten Objekts mit einem Schlüssel.

Arrays wurden bereits in Kapitel 3 behandelt. Ich gehe deshalb hier nur noch auf Auflistungen ein. Alle Auflistungen werden im Namensraum `System.Collection` verwaltet, einige spezielle finden Sie im Namensraum `System.Collection.Specialized`. Die Anzahl und Mächtigkeit der Auflistungen ist so groß, dass ich hier nur einen kleinen Ausschnitt beschreiben kann. Das .NET-Framework stellt für fast jede denkbare Anwendung eine Auflistung zur Verfügung. Es gibt sogar eine, die für die Speicherung von maximal zehn (!) Objekten optimiert ist (`ListDictionary`). Wow.

5.6.1 Die Schnittstellen der Auflistungen

Sorry, jetzt wird es ein wenig kompliziert: Alle Auflistungen implementieren die Schnittstelle `ICollection`. `ICollection` selbst ist von `IEnumerable` abgeleitet. Deshalb besitzen alle Auflistungen die Eigenschaften und Methoden dieser Schnittstellen. Die wichtigste `ICollection`-Eigenschaft ist `Count`, über die Sie die Anzahl der gespeicherten Objekte ermitteln können. Die `GetEnumerator`-Methode der `IEnumerable`-Schnittstelle gibt bei Auflistungen, die (z. B. mit `foreach`) durchgehbar sein sollen, eine Referenz auf die `IEnumerator`-Schnittstelle zurück, die alle diese Auflistungen ebenfalls implementieren. Über die Methoden (`MoveNext`, `Reset`) und die Eigenschaft (`Current`) dieser Schnittstelle können Sie eine Auflistung durchgehen und auf die Elemente zugreifen. Das will ich hier nicht erläutern, weil Sie in C# auch mit `foreach` durchgehen können. `foreach` nutzt aber natürlich die Elemente von `IEnumerator`.

Auflistungen, die sortierbar sind und einen Zugriff auf die Elemente über einen Index erlauben, implementieren zusätzlich die Schnittstelle `IList`. Auflistungen, die die Objekte über einen Schlüssel verwalten, implementieren weiterhin die Schnittstelle `IDictionaryEnume-`

rator. Tabelle 5.13 zeigt die wichtigsten Eigenschaften und Methoden von IList, Tabelle 5.14 die Eigenschaften von IDictionaryEnumerator.

Eigenschaft / Methode	Beschreibung
`object this[int index]`	Über diesen Indizierer erhalten Sie Zugriff auf ein Element.
`int Add(object value)`	fügt ein Element an die Liste an. Gibt den Index zurück, an dem das Element angefügt wurde.
`void Clear()`	entfernt alle Elemente aus der Liste
`bool Contains(object value)`	ermittelt, ob ein Element mit einem angegebenen Wert existiert
`int IndexOf(object value)`	ermittelt den Index eines Elements mit einem bestimmten Wert
`void Insert(int index, object value)`	fügt ein Element in die Liste ein
`void Remove(object value)`	löscht das erste Element, das den angegebenen Wert besitzt
`void RemoveAt(int index)`	löscht das Element, das am angegebenen Index gespeichert wird

Tabelle 5.13: Die wichtigsten Eigenschaften und Methoden der IList-Schnittstelle

Eigenschaft	Beschreibung
`Entry`	gibt ein `DictionaryEntry`-Objekt zurück, das im Wesentlichen den Schlüssel und den Wert eines Elements speichert.
`Key`	gibt den Schlüssel des aktuellen Elements zurück
`Value`	gibt den Wert des aktuellen Elements zurück

Tabelle 5.14: Die Eigenschaften der IDictionaryEnumerator-Schnittstelle

5.6.2 Die wichtigsten Auflistungen

In den Basisklassen des .NET-Frameworks finden Sie viele fertige Klassen zur Erzeugung von Auflistungen und zwei Klassen zur Definition eigener Auflistungen.

System.Collection.ArrayList

Eine Instanz der `ArrayList`-Klasse ist im Prinzip ein Array, das aber dynamisch erweitert werden kann. Beim Erzeugen können Sie eine Startkapazität angeben:

```
ArrayList al = new ArrayList(100);
```

Die Kapazität definiert, wie viele Elemente die Liste zunächst im Speicher reserviert. Wenn Sie mehr Elemente hinzufügen, als verfügbar sind, wird die Liste einfach implizit um die Kapazität erweitert. Die Voreinstellung der Kapazität beträgt 16. Anfügen können Sie beliebige Typen über die `Add`-Methode, einfügen können Sie über die `Insert`-Methode:

```
al.Add("Ford"); // anhängen
al.Add("Arthur");
al.Insert(0, "Zaphod"); // am Anfang einfügen
```

Durchgehen können Sie die Liste mit `foreach`:

```
foreach (string name in al)
   Console.WriteLine(name);
```

oder mit `for`:

```
for (int i=0;i<al.Count;i++)
   Console.WriteLine(al[i]);
```

`for` ist in vielen Fällen (und bei allen Auflistungen) schneller als `foreach`, weil dazu kein Enumerator erforderlich ist.

Zugriff auf die Elemente erhalten Sie über den Indizierer der `IList`-Schnittstelle:

```
Console.WriteLine(al[0]); // erstes Element auslesen
```

Weitere wichtige Methoden sind `AddRange` (Hinzufügen der Elemente einer anderen Auflistung), `Sort` (Sortieren) und `BinarySearch` (in der

sortierten Liste binär suchen). `Sort` und `BinarySearch` arbeiten wie beim Array (Kapitel 3).

System.Collections.Hashtable

Eine `Hashtable` ist eine Auflistung, die ihre Elemente mit einem Schlüssel assoziiert. Den Schlüssel geben Sie beim Hinzufügen von Elementen an. Er kann einen beliebigen Datentyp besitzen. Üblich sind `int`- oder `string`-Schlüssel. Die `Hashtable` heißt übrigens so, weil sie nicht die Schlüssel selbst, sondern den Hashcode der Schlüssel speichert. Ein Hashcode ist eine über einen komplexen Algorithmus aus einem Wert errechnete Integer-Zahl, die den Wert eindeutig identifiziert. Alle Typen besitzen die von `object` geerbte `GetHashCode`-Methode, die einen Hashcode zurückgibt und die in vielen Typen mit einer speziellen Implementierung überschrieben ist. Beim Zugriff auf die Elemente wird aus dem Schlüssel wieder der Hashcode errechnet und mit den gespeicherten verglichen.

Das folgende Beispiel erzeugt ein `Hashtable`-Objekt und fügt einige Strings mit einem `int`-Schlüssel an:


```
Hashtable ht = new Hashtable(100);
ht.Add(1001,"Zaphod");
ht.Add(1002,"Arthur");
ht.Add(1003,"Ford");
ht.Add(1004,"Trillian");
```

Sie können Elemente aber auch hinzufügen, indem Sie einfach darauf zugreifen. Die `Hashtable`-Klasse besitzt (wie andere ähnliche Klassen auch) ein Feature (das manchmal auch zum Fluch werden kann): Wenn Sie auf ein Element zugreifen, das noch nicht existiert, wird dieses implizit angelegt:

```
ht[1005] = "Marvin"; // Element wird implizit erzeugt
```

Über den Schlüssel können Sie sehr schnell auf einzelne Elemente zugreifen:

```
Console.WriteLine(ht[1002]);
```

Beachten Sie, dass ein Schlüssel nur einmal vorkommen darf. Wenn Sie versuchen ein Element mit einem Schlüssel anzufügen, der bereits existiert, erzeugt die CLR eine Ausnahme vom Typ `ArgumentException`.

Achten Sie gut auf den Datentyp des Schlüssels. Ein `Hashtable`*-Objekt erlaubt (wie auch die Instanzen anderer Klassen, die Schlüssel verwalten), dass Sie Elemente mit prinzipiell gleichwertigen Schlüsseln anfügen, die einen unterschiedlichen Datentyp besitzen. So können Sie z. B. ein Element mit dem Schlüssel 1000 und ein Element mit dem Schlüssel "1000" anfügen (weil 1000 einen anderen Hashcode ergibt als "1000"). Achten Sie auch beim Zugriff auf den Datentyp. Wenn Sie die Elemente beispielsweise mit einen* `string`*-Schlüssel angefügt haben und später über einen* `int`*-Schlüsselwert darauf zugreifen wollen, erhalten Sie ein leeres Element zurück.*

Eine `Hashtable`-Auflistung können Sie nur mit `foreach` durchgehen. `for` funktioniert nicht, weil Sie keinen Integer-Index verwenden können. Sie können die Schlüssel und die Werte separat durchgehen. Deswegen müssen Sie auch entweder die `Values`- oder die `Keys`-Eigenschaft angeben:

```
foreach (string name in ht.Values)
   Console.WriteLine(name);
```

Wenn Sie eine `Hashtable`*-Auflistung sequenziell durchgehen, entspricht die Reihenfolge nicht der, mit der Sie die Elemente angefügt haben. Das kann manchmal zum Problem werden. In meinen Tests entsprach die Reihenfolge beim Durchgehen der Werte immer der umgekehrten Reihenfolge der Schlüssel. Aber darauf würde ich lieber nicht zählen ...*

Die anderen wichtigen Auflistungen

Weil ich im Buch noch andere wichtige Techniken wie den Windows-Programmierung und den Datenzugriff besprechen will, bleibt mir nicht genug Platz, alle wichtigen Auflistungen zu beschreiben. Deswegen liste ich die wichtigsten in Tabelle 5.15 auf.

Auflistung	Beschreibung
NameValue-Collection	Diese im Namensraum System.Collections.Specialized verwaltete Auflistung speichert ähnlich einer Hashtable Werte. Die Werte und die Schlüssel sind allerdings Strings. Sie können über einen Integer-Index auf die Werte zugreifen oder über den Schlüssel. Diese Auflistung ist auf die Speicherung von Stringpaaren optimiert. Gehen Sie die Auflistung idealerweise mit for durch.
Queue	Diese Auflistung repräsentiert eine First-In-First-Out-Liste. Über die Enqueue-Methode können Sie Elemente hinzufügen, über Dequeue lesen Sie die Elemente der Reihe nach wieder aus.
SortedList	Eine Instanz der SortedList-Klasse speichert ähnlich einer Hashtable Elemente, die über einen Schlüssel identifiziert werden. Sie Elemente werden allerdings automatisch nach dem Schlüssel sortiert. Sie können über den Schlüssel oder über einen Integer-Index auf die Elemente. Für den Zugriff über einen Integer-Index müssen Sie allerdings die GetByIndex-Methode verwenden.
Stack	Diese Auflistung repräsentiert eine First-In-Last-Out-Liste. Sie können über die Push-Methode Elemente anfügen. Mit Peek können Sie das unterste Element auslesen ohne dies zu löschen. Pop liest das unterste Element aus und löscht dieses gleich.

Tabelle 5.15: Die anderen wichtigen Auflistungen

5.6.3 Typsichere eigene Auflistungen

Die Auflistungen des .NET-Framework haben einen Nachteil, wenn Sie diese verwenden, um Objekte zu speichern: Sie sind nicht typsicher. Das liegt ganz einfach daran, dass der gespeicherte Datentyp object ist. Natürlich können Sie diese Auflistungen für das Speichern von Objekten verwenden, aber Sie können nie sicher sein, dass auch wirklich Ihre Objekte gespeichert sind. Außerdem müssen Sie beim Lesen immer casten, um auf die Eigenschaften und Methoden der Objekte zugreifen zu können. Viel eleganter ist eine eigene, typsi-

chere Auflistung. Ich zeige hier einfach anhand eines erläuterten, typischen Beispiels, wie Sie eine solche erzeugen und verwenden. Die Auflistung soll Objekte der Klasse *Person* verwalten. Die Klasse wird dazu von der Klasse `DictionaryBase` abgeleitet, die die Basiselemente zur Verfügung stellt. Ein `Dictionary` ist übrigens eigentlich eine `Hashtable`[3]. Warum Microsoft die Klasse nicht HashtableBase genannt hat, weiß nur der Wind ...

Die *Person*-Klasse implementiert die wichtigen Methoden `ToString` und `Equals`. Den Grund dafür beschreiben die Kommentare:

```
/* Klasse zur Speicherung von Personendaten */
public class Person
{
   public int ID;
   public string FirstName;
   public string SurName;

   /* Der Konstruktor */
   public Person(int id, string firstName,
      string surName)
   {
      this.ID = id;
      this.FirstName = firstName;
      this.SurName = surName;
   }

    /* Eine Eigenschaft für den vollen Namen */
   public string FullName {get {return FirstName +
      " " + SurName;}}

   /* Die ToString-Methode wird überschrieben,
    * u. a. damit bei der Datenanbindung von Arrays
    * oder Collections dieses Typs an Steuerelemente
```

3. Im eigentlichen Sinn ist ein Dictionary eine Auflistung, die String-Paare speichert. Unter Visual Basic 6 war es aber schon immer möglich, beliebige Werte/Objekte in einem Dictionary mit einem beliebigen Schlüssel zu speichern. Also eigentlich genau das, was eine Hashtable macht.

```
     * ein passender Wert ausgegeben wird. */
    public override string ToString()
    {
       return (ID.ToString() + ": " + FirstName +
          " " + SurName);
    }

    /* Die Equals-Methode wird überschrieben. Diese
     * Methode wird bei Vergleichen zweier Objekte
     * dieser Klasse verwendet. Unter anderem ruft die
     * IndexOf-Methode einer Collection diese Methode
     * auf */
    public override bool Equals(object obj)
    {
       Person p = (Person)obj;
       if (this.ID == p.ID)
          /* zwei Objekte mit gleicher ID
           * werden als gleich gewertet */
          return true;
       else
          return false;
    }
}
```


Eine Klasse, die Personen-Objekte auflisten soll, wird nun von `DictionaryBase` abgeleitet, damit sie die grundlegenden Elemente erbt:

```
using System;
using System.Collections;

namespace Nitty.Gritty.Samples.TypsichereAuflistung
{
   public class PersonDictionary: DictionaryBase
   {
```

Die Methoden `Add`, `Contains` und `Remove` müssen implementiert werden. Das Beispiel implementiert diese Methoden typsicher: Die Auflistung soll die ID der gespeicherten Personen als Schlüssel verwalten. Der `Add`-Methode muss dieser Schlüssel allerdings nicht

übergeben werden, weil das übergebene *Person*-Objekt den Schlüssel ja bereits speichert:

```
public Person Add(Person p)
{
   Dictionary.Add(p.ID, p);
   return p;
}

public bool Contains(int key)
{
   return Dictionary.Contains(key);
}

public void Remove(int id)
{
   Dictionary.Remove(id);
}
```

Wie Sie dem Beispiel entnehmen können, greifen Sie über die (geerbte) `Dictionary`-Eigenschaft auf die zugrunde liegende Auflistung zu.

Um die Auflistung später mit `foreach` durchgehen zu können, sollten Sie die `Values`- und die `Keys`-Eigenschaft implementieren:

```
/* Die schreibgeschützte Values-Eigenschaft */
public System.Collections.ICollection Values
{
   get {return Dictionary.Values;}
}

/* Die schreibgeschützte Keys-Eigenschaft */
public System.Collections.ICollection Keys
{
   get {return Dictionary.Keys;}
}
```

Schließlich ermöglicht noch ein Indizierer den Zugriff auf die Personen:

```
public Person this[int id]
{
```

```
            set {Dictionary[id] = value;}
            get {return (Person)Dictionary[id];}
        }
    }
}
```

Fertig!

Wenn Sie nun mit Instanzen dieser Klasse arbeiten, ist der Umgang damit typsicher und intuitiv:

```
/* Personen-Auflistung erzeugen */
PersonDictionary persons = new PersonDictionary();

/* Einige Personen erzeugen und der Auflistung
 * hinzufügen */
persons.Add(new Person(1001,"Stephen Titus","George"));
persons.Add(new Person(1002,"Aurora",
   "Borealis Smith"));
persons.Add(new Person(1003,"Kalliope",""));
persons.Add(new Person(1004,"Puck",""));
persons.Add(new Person(1005,"Zephyr",""));

/* Auf eine Person über den Schlüssel zugreifen */
int id = 1003;
Console.WriteLine("Die Person mit der ID {0} " +
   "heißt {1}", id, persons[id].FullName);

/* Alle Personen durchgehen */
foreach (Person p in persons.Values)
{
   Console.WriteLine(p.ToString());
}

/* Eine Person löschen (aus der Geschichte entfernen,
 * damit Stephen und Aurora endlich zusammenkommen) */
persons.Remove(1001);
```

6 Ausnahmebehandlung und Debugging

Schon während der Programmentwicklung treten beim Test des Programms immer wieder Ausnahmen (Laufzeitfehler) auf. Wenn Sie z. B. mit ADO eine Verbindung zu einer Datenbank aufbauen wollen und Ihre Verbindungsangaben sind nicht korrekt (was in der Praxis häufig vorkommt, wenigstens bei mir ...) oder die Datenbank wird nicht gefunden, meldet Ihr Programm eine so genannte Ausnahme (Exception). Dieses Kapitel zeigt im ersten Teil, wie Sie Ausnahmen in Ihren eigenen Klassen selbst erzeugen und abfangen.

Da Sie wahrscheinlich (genau wie ich) nicht immer und überall Ausnahmen abfangen, treten beim Testen eines Programms auch unbehandelte oder unerwartete Ausnahmen auf. Außerdem enthalten Programme häufig logische Fehler, die zu einem Fehlverhalten der Anwendung führen. Diese Ausnahmen und Fehler müssen Sie dann debuggen. Ab Seite 291 zeige ich, wie Sie das machen.

In diesem Kapitel werden die Klassen `Debug` *und* `Trace` *aus dem Namensraum* `System.Diagnostics` *nicht behandelt, die erweiterte Debugging-Möglichkeiten bieten. Auf der Website zum Buch finden Sie einen entsprechenden Artikel dazu.*

6.1 Ausnahmebehandlung

Unbehandelte Ausnahmen und die CLR

Ausnahmen können in jedem Programm auftreten. Wenn z. B. eine Textdatei, die geöffnet werden soll, nicht vorhanden ist, tritt eine Ausnahme ein. Bei den Ausnahmen werden zunächst solche unterschieden, die Ihr Programm behandelt (also abfängt) und solche die nicht behandelt werden.

Wenn Sie eine Anwendung (über Visual Studio oder direkt) ausführen, wird eine unbehandelte Ausnahme entweder lediglich in einem einfachen Dialog gemeldet oder führt dazu, dass der Visual Studio-

oder CLR-Debugger (der prinzipiell identisch ist mit dem Debugger von Visual Studio) das Programm anhält und dem Anwender die Möglichkeit bietet, das Programm zu debuggen. Was nun passiert, ist von verschiedenen Faktoren abhängig. Der einfache Dialog wird angezeigt, wenn

- die Anwendung direkt unter dem .NET-Framework in der verteilbaren Version (ohne SDK) ausgeführt wird oder
- die Anwendung unter dem .NET-Framework SDK ausgeführt wird, es sich um eine Windowsanwendung (keine Konsolenanwendung) handelt und die Einstellung JITDEBUGGING im Element SYSTEM.WINDOWS.FORMS der .NET-Konfigurationsdatei *machine.config* bzw. der Anwendungskonfigurationsdatei (vgl. Kapitel 9) auf `false` eingestellt ist.

Der einfache Dialog ist an sich bereits sehr aussagekräftig, wie Abbildung 6.1 zeigt. Das verwendete (Windows-)Programm zur »Generierung« dieser Ausnahme enthält den folgenden Quellcode:

```
private void btnException_Click(object sender,
   System.EventArgs e)
{
   int x = 10, y = 0, z;
   z = x / y; // Ausnahme vom Typ DivideByZeroException
}
```

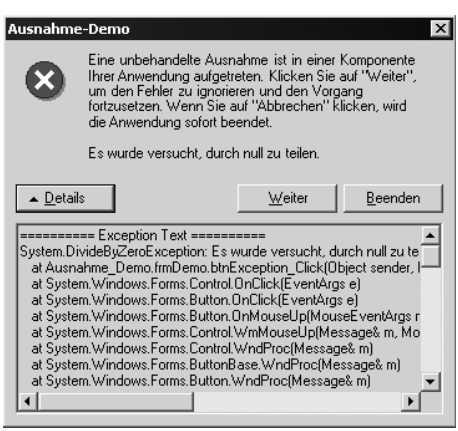

Bild 6.1: Eine Ausnahme wird im Ausnahmedialog gemeldet

Beachten Sie, dass Sie die Einstellung JITDEBUGGING *im Element* SYSTEM.WINDOWS.FORMS *der Datei machine.config oder der Anwendungskonfigurationsdatei auf* `false` *setzen müssen, dass es sich um eine Windows-Anwendung handeln muss, und dass die Ausnahme nicht mit* `double`- *oder* `float`*-Werten erzeugt wird, wenn Sie das einmal selbst ausprobieren wollen. Die Datei machine.config finden Sie im Ordner des .NET-Framework im Unterordner config. Anwendungskonfigurationsdateien werden in Kapitel 9 behandelt.*

Eigentlich ist diese Art der Ausnahmebehandlung für unerwartete Ausnahmen (das sind die, an deren Abfangen Sie beim Programmieren nicht gedacht haben) ganz nett. Der Anwender kann entscheiden, ob er das Programm beenden (womit eventuell nicht gespeicherte Arbeit verloren geht) oder weiterarbeiten will. In den Details findet der Anwender einen Auszug aus der Aufrufreihenfolge die zu dem Fehler geführt hat mit Informationen über die Zeile, in der der Fehler aufgetreten ist. Diese Information ist für unbehandelte Ausnahmen sehr wertvoll für Sie als Programmierer. Damit können Sie auch Fehler ausfindig machen, die nicht auf Ihrem Rechner, aber beim Kunden auftreten.

Sie können potenzielle Ausnahmen aber auch mit einer eigenen Behandlung versehen. Wenn Sie allein dafür sorgen wollen, dass dem Anwender für »unbehandelte« Ausnahmen der BEENDEN-Schalter nicht zur Verfügung steht, müssen Sie alle Ausnahmen abfangen und selbst behandeln. Die Behandlung einer Ausnahme wird in den meisten Fällen die einfache Ausgabe eines Fehlertextes sein. In einigen wenigen Fällen geben Sie dem Anwender vielleicht die Möglichkeit, durch Eingaben die Ausnahmesituation zu bereinigen (z. B. den Dateinamen einer zu öffnenden Datei neu anzugeben).

6.1.1 Grundlagen zu Ausnahmen

Wenn eine Ausnahme eintritt, generiert Ihr Programm im Hintergrund ein Objekt der Klasse `Exception` oder einer davon abgeleiteten Klasse. Die Klasse `Exception` ist die Basisklasse für alle Ausnahmen. Die meisten Klassen des .NET-Framework oder externer Klassenbibliotheken

implementieren aber eigene Klassen, die von `Exception` abgeleitet sind. Tabelle 6.1 zeigt die wichtigsten Ausnahmeklassen des .NET-Framework.

Ausnahmeklasse	Bedeutung
`AccessException`	Fehler beim Zugriff auf eine Eigenschaft oder eine Methode.
`ArgumentException`	Ein Argument einer Methode war ungültig.
`ArgumentNullException`	Einer Methode wurde ein `Null`-Argument übergeben und die Methode akzeptiert dies nicht.
`ArgumentOutOfRange-Exception`	Der Wert eines übergebenen Arguments liegt nicht im gültigen Bereich.
`ArithmeticException`	Ein arithmetische Unter- oder Überlauf ist aufgetreten.
`ArrayTypeMismatch-Exception`	In einem Array wurde versucht einen ungültigen Wert zu speichern.
`DivideByZeroException`	Eine Division durch o ist aufgetreten.
`FormatException`	Beim Versuch, einen Datentyp in einen anderen zu konvertieren oder diesen zu formatieren konnte der Wert nicht konvertiert werden.
`IndexOutOfRange-Exception`	Beim Zugriff auf ein Array wurde ein ungültiger Index verwendet.
`InvalidCastException`	Beim Versuch, einen Datentypen in einen anderen zu casten, ist ein Fehler aufgetreten.
`NullReferenceException`	Es wurde versucht eine Objektreferenz zu verwenden, die kein Objekt referenziert.
`OutOfMemoryException`	Der Speicher reicht für die aktuelle Operation nicht aus.
`StackOverflowException`	Der Stack ist übergelaufen. Dieser Fehler wird meist bei rekursiven Aufrufen von Methoden erzeugt, wenn die Endebedingung der Methode nicht eintritt.

Tabelle 6.1: Die wichtigsten Ausnahmeklassen des .NET-Framework

Eine aufgetretene Ausnahme können Sie in Ihrem Programm abfangen, wobei Sie eine Referenz auf das erzeugte Objekt erhalten können, um damit zu arbeiten. Wie das geht zeige ich ab Seite 279. Zunächst beschreibe ich, wie Sie eigene Ausnahmen erzeugen, damit Sie deren Bedeutung besser verstehen.

6.1.2 Erzeugen eigener Ausnahmen

Ich denke, wenn Sie sehen, wie eine Ausnahme erzeugt wird, erkennen Sie deren Bedeutung besser als mit theoretischen Ausführungen. Die Klasse *Statistics* soll einige statistische Berechnungen auf einer Zahlmenge ermöglichen. Wenn eine statistische Maßzahl abgefragt wird und die Zahlmenge leer ist, soll eine spezielle Ausnahme erzeugt werden. Damit wird sichergestellt, dass der Programmierer, der diese Klasse verwendet, die spezielle Situation nicht ignorieren (und damit u. U. logische Fehler erzeugen) kann. Entweder fängt er die Ausnahme ab oder verlässt sich auf die Ausnahmebehandlung der CLR. Der Anwender erhält aber auf jeden Fall eine Meldung (es sei denn, der Programmierer ignoriert die Ausnahme).

Zunächst erzeugen Sie dazu eine eigene Ausnahme-Klasse, die von der Klasse `Exception` abgeleitet wird.

```
/* Klasse für die spezielle Ausnahme */
public class EmptyNumberArrayException: Exception
{
   /* Konstruktor */
   public EmptyNumberArrayException(
      string message): base(message)
   {
   }
}
```

Dann »werfen« Sie diese Ausnahmen überall da, wo es notwendig ist:

```
/* Klasse zur Berechnung von statistischen Maßzahlen */
public class Statistics
{
   private ArrayList numbers = new ArrayList();
```

```
   /* Methode zum Hinzufügen einer Zahl */
   public void AddNumber(double number)
   {
      numbers.Add(number);
   }

   /* Eigenschaft für die Summe */
   public double Sum
   {
      get
      {
         if (numbers.Count == 0)
            throw new EmptyNumberArrayException(
                    "Cannot calculate sum. " +
                    "The number array is empty.");
         double result = 0;
         for (int i=0; i<numbers.Count; i++)
            result += (double)numbers[i];
         return result;
      }
   }

   /* Eigenschaft für den Mittelwert */
   public double Average
   {
      get
      {
         if (numbers.Count == 0)
             throw new EmptyNumberArrayException(
                "Cannot calculate average. " +
                "The number array is empty.");
         return Sum / numbers.Count;
      }
   }
}
```

So einfach erzeugen Sie eine benutzerdefinierte Ausnahme. Daran erkennen Sie auch, wie die vielen Ausnahme-Klassen des .NET-Frameworks entstanden sind.

Beachten Sie, dass Sie nicht zu viele Ausnahmen in Ihren Klassen erzeugen. Das Werfen und Abfangen von Ausnahmen kostet Performance. Verwenden Sie Ausnahmen nur in wirklichen Ausnahmesituationen und nicht, um dem Programmierer eine Information über einen normalen Zustand zu geben (dazu verwenden Sie besser Rückgabewerte von Methoden). Besonders dann, wenn Sie in einer Klasse eine System-Ausnahme abfangen und in eine eigene Ausnahme weitergeben, wird ein Programm, in dem viele Ausnahmen erzeugt und behandelt werden, ineffizient (weil die CLR dann zwei Ausnahmen behandeln muss). Häufig können Sie sich einfach darauf verlassen, dass die CLR schon selbst eine passende Ausnahme erzeugt.

6.1.3 Abfangen von Ausnahmen

Ausnahmen fangen Sie in `try-catch`-Blöcken ab. Im einfachsten Fall fangen Sie alle Ausnahmen gemeinsam ab, indem Sie die Klasse `Exception` im `catch`-Block angeben:

```
/* Versuch, eine Instanz der Klasse zu verwenden,
 * ohne die Zahlen hinzuzufügen */
stat = new Statistics();
try
{
   Console.WriteLine("Summe: {0}.", stat.Sum);
   Console.WriteLine("Mittelwert: {0}.", stat.Average);
}
catch (Exception ex)
{
   Console.WriteLine(ex.Message);
}
```

Tritt im Programm eine Ausnahme ein, bricht dieses den `try`-Block sofort ab und führt die erste Anweisung im `catch`-Block aus, der die Ausnahme behandelt. Die Variable `ex` (deren Name Sie frei vergeben können) zeigt im `catch`-Block auf das erzeugte `Exception`-Objekt. Sie können die in Tabelle 6.2 aufgelisteten Eigenschaften dieses Objekts verwenden, um den Fehler auszuwerten.

Eigenschaft	Bedeutung
`InnerException`	Wenn eine Ausnahme auf Grund einer anderen Ausnahme geworfen wurde, enthält diese Eigenschaft normalerweise eine Referenz auf die innere Ausnahme.
`Message`	Der Fehlertext der Ausnahme
`Source`	Ein String, der das Objekt oder die Anwendung bezeichnet, die den Fehler erzeugt hat
`StackTrace`	Diese Eigenschaft liefert einen String mit Informationen über die Aufrufliste der Anweisungen die zu der Ausnahme geführt haben. Interessant ist, dass auch die eigentlich fehlerauslösende Zeile inklusive Zeilennummer dort auftaucht.

Tabelle 6.2: Die wichtigsten Eigenschaften der Exception-Klasse

Wenn Sie spezifische Ausnahmen abfangen wollen, können Sie mehrere `catch`-Blöcke für diese Ausnahmen einrichten. Der folgende Quellcode fängt z. B. die möglichen Fehler ab, die bei einem Programm entstehen können, das zwei Ganzzahleingaben multipliziert und das Ergebnis ausgibt:

```
Console.Write("Geben Sie eine Zahl ein: ");
string number1 = Console.ReadLine();
Console.Write("Geben Sie eine weitere Zahl ein: ");
string number2 = Console.ReadLine();

try
{
   Console.WriteLine("Das Ergebnis ist {0}",
      Convert.ToInt32(number1) *
      Convert.ToInt32(number2));
}

/* Die FormatException, die bei der Eingabe ungültiger
 * Zahlwerte auftritt, wird extra abgefangen */
catch (FormatException)
{
```

```
   Console.WriteLine("Sie haben mindestens einen " +
      "ungültigen Zahlwert eingegeben");
}

/* Die OverflowException, die bei der Eingabe zu großer
 * Werte aufgerufen wird, wird ebenfalls extra
 * abgefangen */
catch (OverflowException)
{
   Console.WriteLine("Mindestens einer der " +
      " eingegebenen Werte ist zu groß");
}
/* Alle anderen Ausnahmen werden hier abgefangen */
catch (Exception ex)
{
   Console.WriteLine("Unerwarteter Fehler: " +
      ex.Message + "\n\nWeitere Informationen: " +
      ex.StackTrace);
}
```

Beachten Sie, dass Sie die Option „Auf Unter-/Überlauf prüfen“ in den Eigenschaften des Projekts einschalten müssen, damit bei der Multiplikation der Integerwerte ein Überlauf zu einer Ausnahme führt.

Beim Anfangen der `FormatException`- und `OverflowException`-Ausnahmen verwendet der Beispiel-Quellcode keine Variable für die Ausnahme, weil diese nicht benötigt wird. Er gibt aber die Ausnahmeklasse im `catch`-Block an, damit diese Ausnahme abgefangen wird. Wenn das Programm eine Ausnahme erzeugt, überprüft die CLR, ob in der aktiven Ausnahmebehandlung ein `catch`-Block gefunden wird, dessen Klasse dieselbe ist wie die der Ausnahme. Wird eine solche gefunden, verzweigt die CLR in den betreffenden `catch`-Block. Wird die Klasse nicht gefunden, verzweigt das Programm in den Block, der die Ausnahmen vom Typ `Exception` abfängt (diesem Typ gehören alle Ausnahmen an). Ist dieser Block nicht vorhanden, wird die Ausnahme in der Aufrufliste (siehe Seite 282) nach oben weitergegeben. Findet sich auch dort keine Behandlung, zeigt das Programm die Standard-Ausnahmebehandlung an.

Wenn Sie bereits eine generelle Ausnahmebehandlung (der Klasse `Exception`*) in Ihrem Code eingebettet haben, zeigt Ihnen diese nicht mehr die Klasse des erzeugten Fehlers an. Wenn Sie spezielle Ausnahmen extra behandeln wollen, müssen Sie die Klasse der Ausnahme kennen. Um diese herauszufinden, sollten Sie den Debugger so umstellen, dass dieser auch bei behandelten Ausnahmen anhält. Wie das geht zeige ich ab Seite 301.*

6.1.4 Die Aufrufliste und der Stack-Trace

Wenn eine Ausnahme erzeugt wird, überprüft die CLR, ob in der aktuellen Methode eine Ausnahmebehandlung vorhanden ist. Ist dies der Fall, wird diese daraufhin überprüft, ob die Klasse der erzeugten Ausnahme dort abgefangen wird. Ist keine Behandlung vorhanden oder wird die Klasse nicht abgefangen, wird die Ausnahme in der Aufrufliste nach oben weitergegeben. Das gilt natürlich nur dann, wenn die Ausnahme in einer untergeordnet aufgerufenen Methode erzeugt wird. Die übergeordnete Methode übernimmt dann die Behandlung. Ein (bewusst einfach gehaltenes) Beispiel demonstriert dieses Verhalten:

```
using System;

namespace Nitty_Gritty.Samples.Aufrufliste
{
   class Start
   {
      static double Div(int number1, int number2)
      {
         return number1 / number2;
      }

      static void Calc()
      {
         Console.WriteLine(Div(10,0));
      }

      [STAThread]
```

```
      static void Main(string[] args)
      {

         /* Ausnahmen werden nur auf der
            * obersten Ebene behandelt */
         try
         {
            Calc();
         }
         catch (Exception ex)
         {
            Console.WriteLine("Ausnahme vom Typ {0}: "
               + "{1}\nStack-Trace:\n{2}",
               ex.GetType().Name, ex.Message,
               ex.StackTrace);
         }
      }
   }
}
```

Tritt eine Ausnahme (vom Typ `DivideByZeroException`) in der Methode *Div* auf, wird diese an die Methode *Calc* weitergegeben. Wird diese Ausnahme in *Calc* nicht behandelt, so wird diese an die übergeordnete Methode *Main* weitergegeben und dort behandelt.

Der Stack-Trace

Der Stack-Trace[1] des `Exception`-Objekts beinhaltet Informationen über den Aufrufpfad bei einer eingetretenen Ausnahme. Das Beispielprogramm oben gibt bei einer Ausnahme u. a. den Stack-Trace aus:

```
Ausnahme vom Typ DivideByZeroException: Attempted to divide by zero.
Stack-Trace:
   at Demo.Start.Div(Int32 number1, Int32 number2) in
C:\Demo\Start.cs:line 9
   at Demo.Start.Calc() in C:\Demo\Start.cs:line 14
   at Demo.Start.Main(String[] args) in C:\Demo\Start.cs:line 25
```

1. Trace = Spur

6.1.5 Tipps für die Praxis

Den Stack-Trace bei unerwarteten Ausnahmen ausgeben

Wenn Sie *unerwartete* Ausnahmen selbst behandeln wollen (anstatt sich auf die Standardbehandlung der CLR zu verlassen), geben Sie möglichst immer den Stack-Trace mit aus. Die dort enthaltenen Informationen helfen Ihnen dabei, Ausnahmen, die nur beim Anwender auftreten, zu lokalisieren. In der Praxis treten viele Ausnahmen nicht bereits dann auf, wenn Sie das Programm testen, sondern erst dann, wenn ein Anwender mit dem Programm arbeitet. Das liegt häufig daran, dass die Umgebung des Anwenders anders ist als bei Ihnen oder dass der Anwender einfach mehr ausprobiert als Sie.

Ausnahmebehandlung nicht zu weit treiben

Treiben Sie die Ausnahmebehandlung nicht zu weit. Meist reicht die einfache Ausgabe des originalen (mittlerweile recht aussagekräftigen) Fehlertextes der Ausnahme aus, um dem Anwender mitzuteilen, dass etwas passiert ist. Fangen Sie in untergeordnet aufgerufenen Methoden nur die Ausnahmen ab, die Sie erwarten (z. B. beim Öffnen einer Datei die Ausnahme, dass die Datei nicht vorhanden ist). Fangen Sie in Windowsanwendungen unerwartete Ausnahmen entweder in den Ereignismethoden (die ja immer am Anfang der Aufrufkette stehen) über den Typ `Exception` ab, oder verlassen Sie sich auf die Standardbehandlung für solche Ausnahmen. Da Sie auch in eigenen Behandlungen Informationen über die Aufrufliste erhalten können, ist dieses Vorgehen für die Praxis recht interessant.

6.1.6 Schachteln von Ausnahmebehandlungen

Ausnahmebehandlungen können auch geschachtelt werden. Das ist immer dann wichtig, wenn Sie einzelne Anweisungen unabhängig von einem bereits vorhandenen `try-catch`-Block behandeln wollen. Das Beispielprogramm kann z. B. die einzelnen Zahleingaben separat auswerten:

```
/* Den Anwender Zahlen eingeben lassen */
Console.Write("Geben Sie eine Zahl ein: ");
string input1 = Console.ReadLine();
Console.Write("Geben Sie eine weitere Zahl ein: ");
```

```
string input2 = Console.ReadLine();

/* Variable für die Zahlen und das Ergebnis.
 * Da die Zahlen in einem try-catch-Block zugewiesen
 * werden, muss eine Initialisierung erfolgen, da der
 * Compiler ansonsten den Fehler meldet, dass diese
 * möglicherweise nicht zugewiesen sind */
int number1 = 0, number2 = 0;

/* Eine Auflistung, die Informationen über die
 * aufgetretene Ausnahmen speichert */
ArrayList errors = new ArrayList();

/* Allgemeine Behandlung für unerwartete Ausnahmen */
try
{
   /* Die erste eingegebene Zahl wird auf Gültigkeit
    * überprüft */
   try
   {
      number1 = Convert.ToInt32(input1);
   }
   catch (FormatException)
   {
      errors.Add("Zahl 1 ist keine gültige Zahl.");
   }

   /* Die zweite eingegebene Zahl wird auf Gültigkeit
      * überprüft */
   try
   {
      number2 = Convert.ToInt32(input2);
   }
   catch (FormatException)
   {
      errors.Add("Zahl 2 ist keine gültige Zahl.");
   }
```

```
   /* Überprüfen, ob bei der Auswertung der Eingaben
    * Fehler aufgetreten sind */
   if (errors.Count > 0 )
   {
      /* Die in der ArrayList gespeicherten Strings
       * werden etwas "tricky" über String.Join
       * zusammengefasst und ausgegeben */
      string[] a = new String[errors.Count];
      errors.CopyTo(a,0);
      Console.WriteLine(String.Join("\n",a));
   }
   else
   {
      Console.WriteLine("{0} * {1} = {2}",
         number1, number2, number1 * number2);
   }
}

/* Alle anderen Ausnahmen werden hier abgefangen */
catch (Exception ex)
{
   Console.WriteLine("Unerwarteter Fehler: {0}\n\n" +
      "Weitere Informationen:\n{1}", ex.Message,
      ex.StackTrace);
}
```


Die Meldungen der aufgetretenen Ausnahmen werden (in meinen Anwendungen) häufig in einer Auflistung zwischengespeichert. Diese Auflistung wird vor der Berechnung überprüft. Wenn die Auflistung Werte speichert, ist mindestens ein Fehler aufgetreten. Das Beispiel gibt die Fehlermeldungen dann (über einen kleinen Trick) einfach aus.

6.1.7 Ignorieren von Ausnahmen

In manchen Fällen ist es sinnvoll, Ausnahmen einfach zu ignorieren. Das trifft besonders für Anweisungen zu, die auf jeden Fall ausführen wollen, auch wenn Ausnahmen möglich sind. Dazu verwenden Sie einen einfachen `try-catch`-Block:

```
try
{
   /* Anweisungen, die eine Ausnahme erzeugen können
    * z. B.:/

}
catch
{
   /* Alle Ausnahmen werden einfach ignoriert */
}
```

Der `catch`-Block enthält keine Anweisungen. Tritt eine Ausnahme im `try`-Block ein, wird diese einfach ignoriert.

6.1.8 Der finally-Block

Wenn eine Ausnahme behandelt (oder ignoriert) wird, müssen häufig nach dem Eintritt der Ausnahme spezielle Anweisungen ausgeführt werden.

In einem `try`-Block öffnen und bearbeiten Sie (über eine `StreamWriter`-Instanz) beispielsweise eine Textdatei:

```
string filename = "C:\Demo.txt";
/* Die Variable für den StreamWriter muss außerhalb des
 * try-Blocks deklariert werden, damit diese im
 * finally-Block gültig ist. Um zu verhindern, dass der
 * Compiler meldet, dass (im finally-Block) eine
 * möglicherweise nicht zugewiesene Variable verwendet
 * wird, wird diese mit null vorbelegt. */
System.IO.StreamWriter sw = null;

try
{
   /* Versuch eine Datei zu öffnen und einen Text zu
    * schreiben */
   sw =  new System.IO.StreamWriter(filename, true);
   sw.WriteLine("This i a test");
}

catch (Exception ex)
```

```
{
   Console.WriteLine(ex.Message);
}
```

Wenn eine Ausnahme eintritt, muss nun die eventuell geöffnete Datei auf jeden Fall wieder geschlossen werden (Merksatz: Ressourcen, die Sie öffnen, sollten Sie auch wieder schließen). Dazu steht Ihnen der `finally`-Block zur Verfügung. Alle Anweisungen in diesem Block werden ausgeführt, nachdem eine Ausnahme eingetreten ist. Sie müssen aber beachten, dass dabei auch wieder Ausnahmen auftreten können. Wenn eine Textdatei beispielsweise nicht geöffnet werden konnte, können Sie diese auch nicht schließen. Diese Ausnahmen würden dann an den übergeordneten Block weitergegeben werden. Deshalb sollten Sie die Anweisungen im `finally`-Block normalerweise wieder in einen `try-catch`-Block einbinden, der alle Fehler ignoriert:

```
finally
{
   /* Die Datei wird auf jeden Fall geschlossen, auch
    * dann, wenn ein Fehler auftritt. Da hier aber
    * auch wieder ein Fehler auftreten kann (nämlich
    * dann, wenn die Datei nicht geöffnet werden
    * konnte) werden alle Fehler einfach ignoriert */
   try
   {
      sw.Close();
   }
   catch
   {
      /* Nichts machen */
   }
}
```


6.1.9 Globale Ausnahmebehandlung

Eine der ersten Fragen in meinen Seminaren ist immer, ob es möglich ist, eine globale Ausnahmebehandlung zu erzeugen. Die Antwort ist: Ja. Für behandelte Ausnahmen können Sie eine Klasse implementieren, die die Ausgabe der Informationen übernimmt und die Sie dann

in der Ausnahmebehandlung verwenden. Für unbehandelte Ausnahmen können Sie eine spezielle Klasse erzeugen. Beim Programmstart erzeugen Sie daraus eine Instanz und übergeben diese an die Eigenschaft `ThreadException` des `Application`-Objekts. Das `Application`-Objekt gehört zum Namensraum `System.Windows.Forms` und bietet Zugriff auf Eigenschaften der Anwendung. Sie ersetzen damit die Defaultbehandlung der CLR. Globale Fehlerbehandlung ist in dieser Form aber nur für Windowsanwendungen möglich. Konsolenanwendungen erlauben keine globale Fehlerbehandlung, in Webanwendungen wird die globale Fehlerbehandlung über spezielle Webseiten realisiert.

Für eine globale Ausnahmebehandlung benötigen Sie also eine Klasse, die eine Methode für das Ereignis `OnThreadException` implementiert:

```
namespace Globale_Ausnahmebehandlung
{
   using System;
   using System.Threading;
   using System.Windows.Forms;

   internal class CustomExceptionHandler
   {
       /* Diese Methode wird bei allen unbehandelten
        * Ausnahmen an Stelle der Default-CLR-Methode
        * aufgerufen, wenn diese an das
        * ThreadException-Ereignis des Application-
        * Objekts angebunden wird */
       public void OnThreadException(object sender,
          ThreadExceptionEventArgs e)
       {
```

In dieser Methode können Sie über das Argument *e* und dessen Eigenschaft `Exception` auf die Ausnahme zugreifen. Dabei müssen Sie allerdings vorsichtig sein: Wenn in den Anweisungen in dieser Methode selbst wieder eine Ausnahme eintritt, würde das zu einer Endlos-Ausführung führen. Die Ausnahme würde ja wieder dazu führen, dass das `Application`-Objekt dieselbe Methode aufruft. Bauen Sie die Anweisungen zur Ausgabe der Ausnahmeinformationen also wieder in einen `try`-Block ein:

```
            try
            {
            /* Als Beispiel wird der Fehler nur in einer
             * MessageBox ausgegeben. Sie können aber
             * z. B. hier auch eine E-Mail senden oder den
             * Fehler in einer Textdatei protokollieren */
            if (MessageBox.Show("Eine Ausnahme vom Typ " +
               e.Exception.GetType() +
               " ist aufgetreten: " +
               e.Exception.Message + "\n\nStack-Trace: " +
               e.Exception.StackTrace + "\n\nWeiter?",
               "Anwendungs-Ausnahme",
               MessageBoxButtons.YesNo,
               MessageBoxIcon.Error) == DialogResult.No)
            {
               Application.Exit();
            }

            }
            /* Wenn bei der Ausgabe der MessageBox eine
             * Ausnahme eintritt, muss dies als fataler
             * Fehler behandelt werden */
            catch (Exception ex)
            {
               /* Ignorieren weiterer Ausnahmen */
               try
               {
                  MessageBox.Show("Fataler Fehler",
                     ex.Message, MessageBoxButtons.OK,
                     MessageBoxIcon.Error);
               }
               finally
               {
                  /* Anwendung beenden */
                  Application.Exit();
               }
            }
         }
      }
   }
}
```

Das Beispiel verwendet bereits eine MessageBox, die erst im nächsten Kapitel behandelt wird. Die MessageBox kennen Sie alle: Wenn Sie in einem Programm z. B. eine Datei bearbeiten und das Programm schließen, werden Sie normalerweise in einer MessageBox gefragt, ob Sie die Änderungen speichern sollen. Im Beispiel meldet die MessageBox die aufgetretene Ausnahme und fragt den Anwender, ob er weitermachen will. Will der Anwender nicht weitermachen, wird das Programm beendet.

In der Startmethode der Anwendung erzeugen Sie nun eine Instanz dieser Klasse und übergeben diese an die Eigenschaft `ThreadException` des `Application`-Objekts:

```
[STAThread]
static void Main()
{
    /* Den eigenen globalen Ausnahme-Handler
     * erzeugen und anbinden */
    CustomExceptionHandler eh =
       new CustomExceptionHandler();
    Application.ThreadException +=
      new ThreadExceptionEventHandler(
      eh.OnThreadException);

    Application.Run(new frmMain());
}
```

Beachten Sie, dass das Ganze nur in Windows-Anwendungen funktioniert.

6.2 Debuggen

Das .NET-Framework-SDK und Visual Studio enthalten je einen sehr guten Debugger, den Sie nutzen können, um unerwartete Ausnahmen oder logische Fehler in Ihren Programmen zu finden und zu beseitigen. Beide Debugger basieren mit ziemlicher Sicherheit auf derselben Komponente, weil sie identische Funktionalität bieten. Der Debugger ist ziemlich mächtig und kann nicht nur .NET-Anwendungen, sondern auch »alte« Windows-Anwendungen und Scriptprogramme debuggen. Die folgenden Seiten zeigen, wie Sie den Debugger nutzen.

6.2.1 Voraussetzungen

Die wichtigste Voraussetzung dafür, dass Sie Ihren Programmcode debuggen können ist, dass dieser mit Debuginformationen kompiliert wird. Ohne diese Informationen kann der Debugger aus dem erzeugten IL-Code der Assemblierung nicht die passende Stelle im Quellcode des Programms identifizieren. Enthält eine Assemblierung keine Debuginformationen, können Sie zwar auch debuggen, dann müssen Sie sich aber mit dem disassemblierten IL-Code beschäftigen. Abbildung 6.2 zeigt, wie der Debugger dann eine Ausnahme anzeigt, die durch den folgenden Quellcode verursacht wurde:

```
/* Der folgende Quellcode erzeugt eine
   * Ausnahme */
int x = 10, y = 0, result;
result = x / y;
```

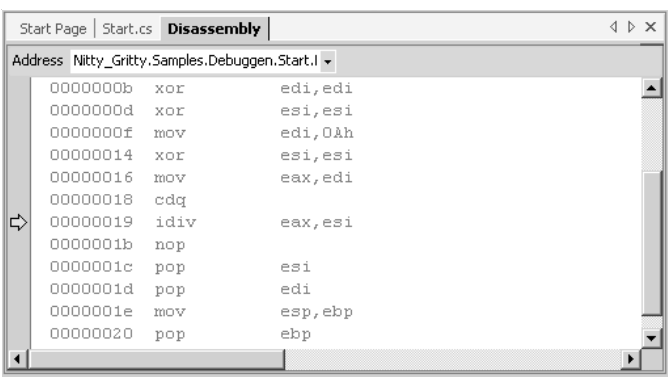

Bild 6.2: Anzeige einer Ausnahme im Debugger wenn keine Debuginformationen in der Assemblierung enthalten sind

Wenn Sie viel Zeit haben und sich mit der komplexen Assembler-Sprache der CLR auseinander setzen wollen, reicht Ihnen diese Anzeige vielleicht aus. Ich verwende lieber die Quellcodeansicht, die ich automatisch dann erhalte, wenn die Assemblierung Debuginformationen enthält und der Quellcode verfügbar ist. Stellen Sie in den Projekteigenschaften für die Konfiguration *Debug* also ein, dass Debuginformationen mit in die Assemblierung kompiliert werden. Bei der

Voreinstellung ist das bereits der Fall. Natürlich müssen Sie Ihre Anwendung dann auch mit dieser Konfiguration kompilieren.

6.2.2 Anhalten des Programms

Um den Debugger nutzen zu können, muss das Programm angehalten werden. Der Debugger hält eine Anwendung automatisch an, wenn eine unbehandelte Ausnahme eintritt und in der .NET-Systemkonfiguration nicht festgelegt ist, dass der Debugger nicht starten soll (siehe Seite 273). Sie können jedoch auch erreichen, dass das Programm auch bei behandelten Ausnahmen anhält. Wie Sie das machen, beschreibe ich allerdings erst ab Seite 301. Für das Lokalisieren logischer Fehler, die nicht zu einer Ausnahme führen, können Sie flexible Haltepunkte setzen. Die einfachste Form eines Haltepunkts, den unbedingten Haltepunkt, beschreibe ich im Folgenden. Bedingte Haltepunkte werden ab Seite 299 behandelt.

Unbehandelte Ausnahmen

Tritt bei der Ausführung eines Programms eine unbehandelte Ausnahme ein, wird mehr oder weniger automatisch der Debugger gestartet. Haben Sie das Programm in Visual Studio gestartet, wird der Fehler automatisch angezeigt. Haben Sie das Programm direkt gestartet, erhalten Sie u. U. vorher die Möglichkeit, einen der installierten Debugger auszuwählen. Nachdem Sie den Debugger ausgewählt haben, öffnet sich dann eventuell noch ein Dialog, in dem Sie die Quellcodedatei suchen müssen. Der Compiler hat zwar den kompletten Pfad zur Quellcodedatei in die Assemblierung integriert. Wenn diese sich aber bei der Ausführung der Anwendung an einem anderen Ort befindet, müssen Sie den Pfad eben angeben. In Visual Studio ist das Ganze nicht notwendig, weil das Programm ja komplett innerhalb der Entwicklungsumgebung ausgeführt wird.

Schließlich öffnet der Debugger die Quellcodedatei und gibt Ihnen die Möglichkeit, das Programm zu debuggen (was Sie ja eigentlich wollen) oder zu beenden (Abbildung 6.3).

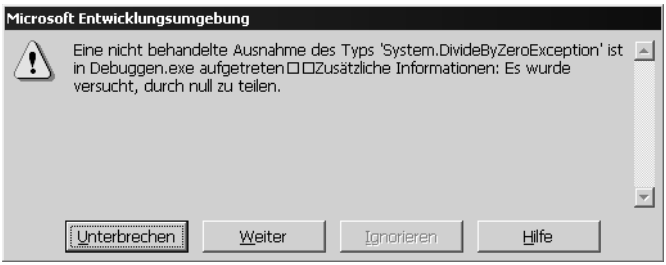

Bild 6.3: Der Debugger hat ein Programm bei einer Ausnahme angehalten und bietet an, das Programm zu unterbrechen

Wenn Sie das Programm unterbrechen, zeigt der Debugger den Quellcode an und markiert die fehlerhafte Anweisung oder die darauf folgende Anweisung (Abbildung 6.4).

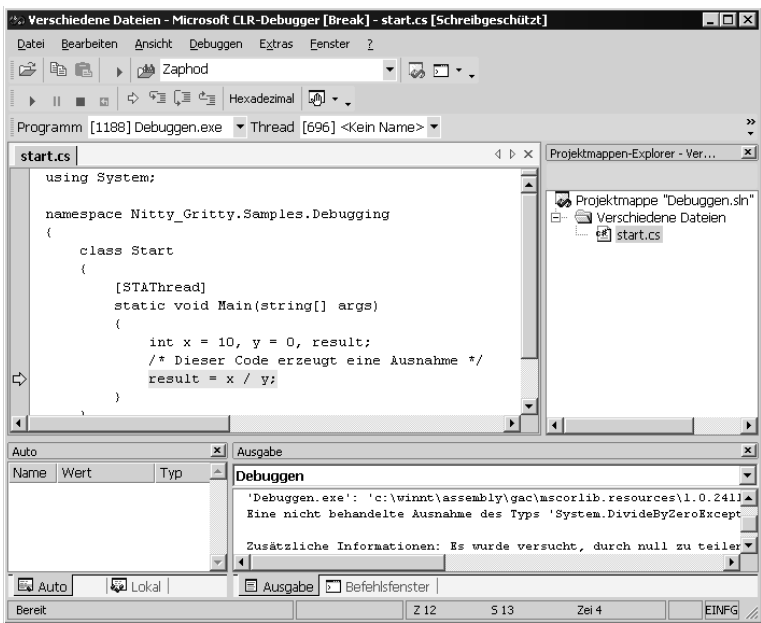

Bild 6.4: Anzeige einer Ausnahme im CLR-Debugger

Beachten Sie, dass der Debugger nicht immer die Anweisung markiert, die den Fehler verursacht hat. In einigen Fällen wird auch die Anweisung markiert, die der ausnahmeerzeugenden Anweisung folgt. Die Markierung sieht dann aber anders aus (grün statt gelb und mit einem anderen Pfeil), wie Abbildung 6.5 zeigt. Ich vermute, dass der Debugger dann an der ausnahmeerzeugenden Anweisung anhält, wenn es sich um einen Ausdruck handelt, der die Ausnahme direkt im Programm erzeugt. Wird die Ausnahme allerdings innerhalb einer Methode erzeugt, die nicht im Quellcode vorliegt, wird die folgende Zeile markiert.

Abbildung 6.5 zeigt ein Beispiel für eine Situation, bei der der Debugger die Anweisung nach der ausnahmeerzeugenden markiert.

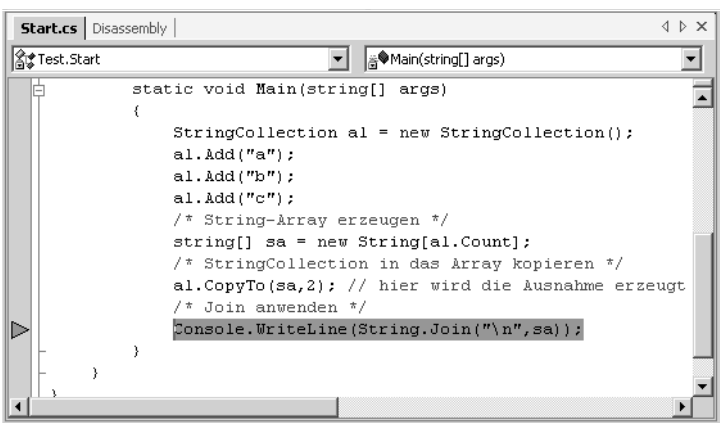

Bild 6.5: Beispiel für eine Ausnahme, bei der der Debugger in der folgenden Zeile anhält

Anhalten »von Hand«

Wenn Sie ein Programm in Visual Studio gestartet haben, können Sie dieses auch mit Strg Pause explizit anhalten. Das ist z. B. immer dann notwendig, wenn das Programm eine Endlosschleife ausführt.

Haltepunkte in Visual Studio

Wenn Sie logische Fehler mit einem Haltepunkt debuggen wollen, ist das Setzen in Visual Studio sehr einfach: Stellen Sie den Eingabecursor in die Anweisung, an der das Programm angehalten werden soll und betätigen Sie F9. Die Anweisung wird rot markiert. Wenn Sie das Programm dann ausführen, hält der Debugger an dieser Anweisung an.

Haltepunkte mit dem CLR-Debugger

Wenn Sie Haltepunkte über den CLR-Debugger setzen wollen, müssen Sie etwas anders vorgehen. Ich beschreibe die Vorgehensweise hier nur kurz, ohne Abbildungen, weil ich noch Platz für weitere Themen benötige.

Starten Sie den CLR-Debugger, den Sie unter dem Namen *DbgCLR.exe* im Ordner *\Programme\Microsoft.NET\FrameworkSDK\GuiDebug* finden (wenn Sie Visual Studio installiert haben, finden Sie diesen Debugger im Ordner *\Programme\Microsoft Visual Studio .NET\FrameworkSDK\GuiDebug*). Öffnen Sie dort die zu debuggende Quellcodedatei (. Setzen Sie Haltepunkte über F9. Nun müssen Sie noch angeben, welche Anwendung ausgeführt werden soll, wenn Sie das Debuggen starten. Wählen Sie dazu den Menübefehl DEBUGGEN / ZU DEBUGGENDES PROGRAMM (DEBUG / PROGRAM TO DEBUG). Suchen Sie dann noch die *exe*-Datei der Anwendung, bestätigen Sie den Dialog und starten Sie schließlich das Debuggen mit F5. Der Debugger hält das Programm am ersten erreichten Haltepunkt an.

6.2.3 Debugging

Wenn das Programm irgendwie angehalten wurde, können Sie mit F11 eine Anweisung weitergehen. Der Debugger verzweigt dann in Methoden, Funktionen und Prozeduren, die in den zu testenden Anweisungen aufgerufen werden. Wollen Sie diese überspringen, betätigen Sie stattdessen F10. Mit ⇧ F11 wird die aktuelle Methode zu Ende ausgeführt. Der Debugger verzweigt dann in die übergeordnete Methode, falls es eine solche gibt. Das ist recht hilfreich, wenn Sie gerade in einer untergeordneten Methode debuggen und in die übergeordnete springen wollen. Mit F5 führen Sie das Programm

ab der aktuellen Anweisung weiter aus. Falls noch ein Haltepunkt folgt, hält der Debugger natürlich dort wieder an. Mit ⇧ F5 beenden Sie das Debuggen (und das Programm).

Die einfachste Art zu debuggen ist nun, den Mauscursor auf einen Bezeichner im Quellcode zu bewegen. Handelt es sich dabei um eine Variable oder Eigenschaft, zeigt der Debugger deren Wert in einem Tooltipp an. Alternativ können Sie auch den Eingabecursor im Quellcode auf eine Variable oder Eigenschaft setzen und ⇧ F9 betätigen, um den Wert der Variablen bzw. Eigenschaft im Schnellüberwachungs-Fenster anzuzeigen.

Sie können aber auch eines der vielen speziellen Debugfenster zur Auswertung von Werten und Ausdrücken verwenden.

Bearbeiten und Fortfahren

Bearbeiten und Fortfahren (Edit and Continue) ist ein nettes Feature des Debuggers, das aber leider nicht so richtig mit C# (und Visual Basic.NET) funktioniert (vgl. die .NET-Dokumentation). Mit diesem Feature können Sie während des Debuggens den Quellcode bearbeiten und das Programm mit dem geänderten Code weiter ausführen. In C# (und Visual Basic.NET) können Sie den Quellcode zwar verändern, wenn Sie diesen dann aber (mit F5) weiter ausführen, zeigt der Debugger in einem Dialog an, dass die Änderungen nicht übernommen werden können.

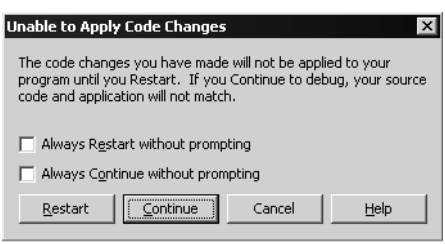

Bild 6.6: Der Dialog, der nach einer Code-Änderung angezeigt wird (englische Version, weil die mir vorliegende ältere deutsche Version nicht identisch ist)

Wenn Sie fortfahren, werden Ihre Code-Änderungen nicht in das kompilierte Programm übernommen (das ist auch irgendwie lo-

gisch). Das Programm läuft also mit dem alten Code weiter. Alternativ können Sie das Programm natürlich neu starten. Dann geht aber der aktuelle Debugging-Status verloren und Sie müssen von vorne beginnen. Das kann in der Praxis sehr nervig sein. Nur damit Sie verstehen, warum ich so kritisch bin: Unter Visual Basic 6 funktioniert dieses Feature einwandfrei (weil der VB6-Debugger den Code interpretiert).

Ich vermute, Bearbeiten und Fortfahren funktioniert nur mit (interpretierten) Script-Programmen korrekt.

Das Befehlsfenster

Wenn das Programm angehalten ist, können Sie im Befehlsfenster (Menü *Debuggen* / *Fenster* / *Direkt*) beliebige Ausdrücke testen. Geben Sie dazu ein Fragezeichen gefolgt vom Ausdruck ein und betätigen Sie ↵. Das Direktfenster gibt das Ergebnis des Ausdrucks aus.

Das Auto- und das Lokalfenster

Das Auto-Fenster (*Debuggen* / *Fenster* / *Auto*) zeigt beim Debuggen automatisch die Werte aller Variablen an, die in der aktuellen Anweisung verwendet werden. Das Lokalfenster (*Debuggen* / *Fenster* / *Lokal*) zeigt die Werte von Variablen an, die in der aktuellen Methode deklariert sind (also lokal gelten).

Das This-Fenster

Das This-Fenster (*Debuggen* / *Fenster* / *This*) zeigt alle Eigenschaften eines Objekts an, in dessen Klasse Sie gerade debuggen. Dieses Fenster funktioniert nur für Instanzen einer Klasse und deshalb z. B. nicht in der (statischen) Startmethode einer Konsolenanwendung.

Die Aufrufliste

In der Aufrufliste (*Debuggen* / *Fenster* / *Aufrufliste*) können Sie alle Aufrufe zurückverfolgen, die zur aktuellen Debugposition geführt haben. Der letzte Aufruf steht immer oben. Über einen Doppelklick auf einem der Aufrufe in der Liste gelangen Sie zu der entsprechenden Stelle im Quellcode. Die Verwendung der Aufrufliste ist manchmal sinnvoll, wenn ein Fehler aufgetreten ist, dessen Ursache eventuell irgendwo im Aufrufpfad liegen könnte.

Das Haltepunkte-Fenster

Das Haltepunkte-Fenster zeigt alle im Projekt gesetzten Haltepunkte an. Über dieses Fenster können Sie Haltepunkte löschen und neue Haltepunkte setzen, die zudem bedingt definiert werden können.

6.2.4 Bedingte Haltepunkte

Für schwieriger zu lokalisierende Fehler können Sie bedingte Haltepunkte definieren. Diese speziellen Haltepunkte besitzen eine Bedingung, die wahr werden muss, damit der Debugger anhält. Außerdem können Sie noch festlegen, wie oft diese Bedingung erfüllt sein muss, damit der Haltepunkt aktiv wird.

Bedingte Haltepunkte sind enorm hilfreich, wenn an einer Anweisung ein logischer Fehler eintritt, der durch einen fehlerhaften Wert in einer Variablen oder Eigenschaft verursacht wird, deren Wert in einem anderen Programmteil verändert wurde. In größeren Programmen wissen Sie in der Regel nicht mehr, welche Programmteile welche Variablen oder Eigenschaften verändern (es sei denn, Sie sind ein Genie). Setzen Sie dann einfach einen bedingten Haltepunkt, der den Wert dieser Eigenschaft bzw. Variablen überwacht und anhält, wenn der Wert eine bestimmte Bedingung erfüllt.

Setzen Sie dazu einen Haltepunkt und öffnen Sie dessen Eigenschaftendialog (Rechtsklick auf dem HALTEPUNKT / HALTEPUNKTEIGENSCHAFTEN).

In den Eigenschaften können Sie Haltepunkte für eine Funktion (Methode) oder Datei setzen. Das macht aber eigentlich gar keinen Unterschied: Bei einem Haltepunkt für eine Funktion wird die Zeile ab dem Funktionsbeginn gezählt, bei einem Haltepunkt für die Datei wird ab dem Dateianfang gezählt. Haltepunkte für Adressen beziehen sich übrigens auf die Speicheradresse. Solche Haltepunkte sind interessant, wenn Sie ein Programm debuggen, dessen Quellcode Sie nicht besitzen.

Über den Schalter BEDINGUNG können Sie die Bedingung definieren. Über den Schalter TREFFERANZAHL können Sie die Anzahl der Treffer einstellen.

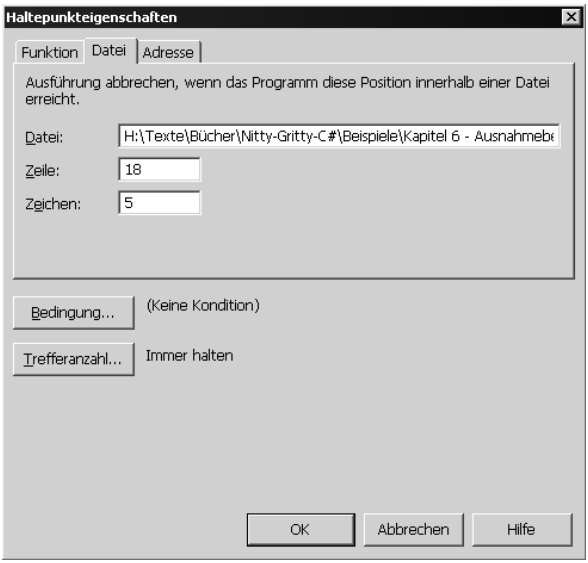

Bild 6.7: Die Eigenschaften eines Haltepunkts

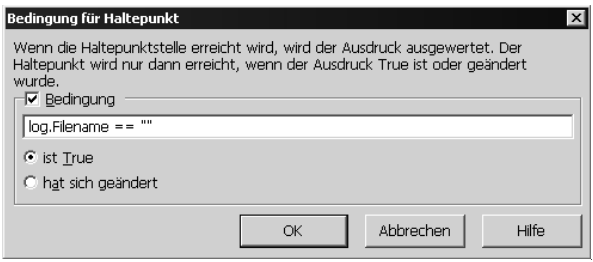

Bild 6.8: Der Dialog zur Einstellung der Bedingung(en) für einen Haltepunkt

Wenn Sie eine Bedingung eingeben, muss diese immer komplett formuliert werden. Wenn Sie eine Variable *i* daraufhin überwachen, ob deren Wert 0 wird, können Sie nicht einfach »== 0« eingeben, sondern müssen komplett »i == 0« schreiben. Interessant ist auch die Option HAT SICH GEÄNDERT. Über diese Option können Sie Haltepunkte setzen, die immer dann zum Anhalten führen, wenn das Ergebnis eines Bedingungsausdrucks sich geändert hat (weil eine Variable oder

Eigenschaft, die im Ausdruck verwendet wird, durch einen Programmteil verändert wurde).

6.2.5 Debuggen bei vorhandener Ausnahmebehandlung

Wenn ein Programmcode, in dem eine Ausnahme auftritt, in einem `try`-Block ausgeführt wird, hält der Debugger (per Voreinstellung) das Programm beim Eintritt der Ausnahme nicht an. Häufig müssen Sie behandelte Ausnahmen aber auch debuggen. Das ist besonders immer dann der Fall, wenn Sie eine Behandlung für unerwartete Ausnahmen vorgesehen haben und eine solche tritt ein.

Schalten Sie dann die Voreinstellung für das Debuggen um. Wählen Sie dazu im Menü DEBUGGEN / FENSTER den Befehl AUSNAHMEN. Im erscheinenden Dialog können Sie einstellen, dass Ihr Programm beim Eintritt einer oder mehrerer Ausnahmen auch dann anhält, wenn diese Ausnahmen behandelt werden.

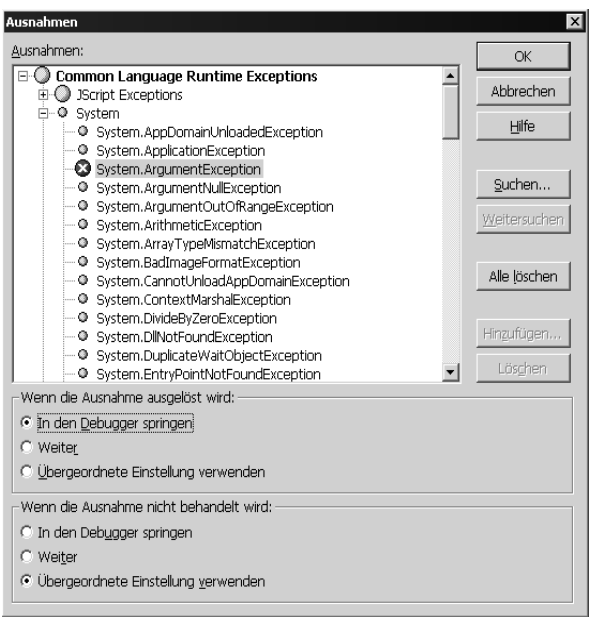

Bild 6.9: Der Dialog zur Einstellung des Verhaltens des Debuggers bei Ausnahmen. Geändert wurde das Verhalten beim Eintritt einer ArgumentException.

Abbildung 6.9 zeigt wie Sie den Debugger so einstellen, dass dieser bei einer Ausnahme vom Typ `ArgumentException` immer anhält. Sie können das Verhalten des Debuggers aber auch für alle Ausnahmen eines Namensraums oder sogar für alle Ausnahmen der CLR anpassen. In der Praxis ist das aber in größeren Projekten meist sehr nervig. Der Debugger hält dann auch wirklich bei jeder Ausnahme an. Auch bei denen, die Sie bewusst ignorieren oder mit einer speziellen Behandlung versehen.

Um den Namen einer Ausnahme herauszubekommen, sollten Sie diesen in allen Ausnahmebehandlungen, die unerwartete Ausnahmen (normalerweise vom Typ `Exception`) behandeln, einfach mit ausgeben. Sie erhalten den Namen über `GeType`:

```
ex.GetType().Name // ex ist die Exception-Variable
```

Die Einstellung im Bereich WENN DIE AUSNAHME NICHT BEHANDELT WIRD *sollten Sie normalerweise nicht ändern. Wenn Sie hier für einzelne oder alle Ausnahmen die Option* WEITER *einstellen, werden Sie beim Testen nicht über eingetretene Ausnahmen informiert.*

Die Einstellung im Bereich WENN DIE AUSNAHME EINTRITT *besitzt eine höhere Priorität als die im Bereich* WENN DIE AUSNAHME NICHT BEHANDELT WIRD. *Wenn Sie für eine Ausnahme z. B. für den Eintritt* IN DEN DEBUGGER SPRINGEN *eingestellt haben und für die Nichtbehandlung* WEITER*, so wird der Debugger auch dann anhalten, wenn die Ausnahme nicht behandelt wird.*

Teil III

Go ahead!

7 Windows-Anwendungen

Dieses Kapitel beschreibt die Programmierung von Windows-Anwendungen, also den Umgang mit Windows-Formularen und den wichtigsten Steuerelementen. Ich zeige dazu zunächst, wie eine einfache Windows-Anwendung im Quellcode aussieht und beschreibe, wie Ereignisse mit Ereignisbehandlungsmethoden verbunden werden (damit Sie das nötige Grundlagenwissen besitzen). Danach zeige ich, wie Sie die .NET-Steuerelemente einsetzen. Eine ausführliche Besprechung der Steuerelemente würde aber den Rahmen dieses Buchs sprengen. Deshalb bespreche ich nur die wichtigsten.

Wenn Sie den Umgang mit den Steuerelementen kennen, sollten Sie noch in der Lage sein, ein Programm zu erzeugen, das mehrere Formulare besitzt. Der letzte Teil dieses Kapitels behandelt deswegen die Arbeit mit Formularen. Hier erfahren Sie dann auch, wie Sie Menüs auf einem Formular erzeugen.

7.0.1 Der Basis-Quellcode einer Windows-Anwendung

Eine Windows-Anwendung startet genau wie eine Konsolenanwendung mit einer statischen Methode `Main`, die in einer Klasse deklariert sein muss. Wenn Sie ein Windows-Anwendung-Projekt mit Visual Studio erzeugen, ist `Main` im Startformular der Anwendung deklariert. Sie können aber auch jede andere Klasse dafür verwenden. Zur besseren Übersicht empfehle ich, die `Main`-Methode in einer eigenen Klasse zu erzeugen, die vielleicht *Start* heißt. Wenn das Projekt mehrere `Main`-Methoden besitzt (was wohl nur sehr selten vorkommt), müssen Sie die Klasse, die die zu verwendende Startmethode enthält in den Projekteigenschaften als Startobjekt angeben. Eine Windows-Anwendung kann übrigens auch, ähnlich einer Konsolenanwendung, ohne Formular starten.

Im Folgenden gehe ich aber davon aus, dass Sie eine normale Windows-Anwendung erzeugen und dass die `Main`-Methode in der Klasse des Startformulars deklariert ist.

Die Klasse des Startformulars wird von der Klasse `System.Windows.Forms.Form` abgeleitet. Damit erbt diese Klasse alle Basis-Ele-

mente und das Basis-Verhalten, das ein Windows-Formular besitzen muss. Eine einfache Windows-Anwendung enthält nur die Deklaration der Klasse:

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

namespace Nitty_Gritty.Samples.Windows-Anwendung
{
   public class StartForm : System.Windows.Forms.Form
   {
      [STAThread]
      static void Main()
      {
         Application.Run(new StartForm());
      }
   }
}
```

Die Anweisung `[STAThread]` ist ein Attribut der `Main`-Methode, das bewirkt, dass diese Methode in einem »Single Thread Apartment« ausgeführt wird. Damit wird sichergestellt, dass nicht mehrere Threads gleichzeitig diese Methode aufrufen können. Threading wird in diesem Buch nicht behandelt.

Die Anweisung `Application.Run(new StartForm());` sorgt dafür, dass die Anwendung startet. `Run` erzeugt dabei eine Nachrichten-Schleife (Message Loop) für den Thread der Anwendung. Diese Schleife wertet die verschiedenen Nachrichten aus, die Windows an die Anwendung sendet (z. B. die Nachricht, dass die Maus betätigt wurde oder dass das aktuelle Fenster geschlossen werden soll). Das übergebene, mit `new` erzeugte Formular, wird als Hauptformular der Anwendung eingesetzt und automatisch geöffnet.

Abbildung 7.1 zeigt das Ergebnis des (mit dem Kommandozeilencompiler) kompilierten Quellcodes. Das erzeugte Formular besitzt (natürlich) keine Steuerelemente, keinen Titel und wird mit einer voreingestellten Größe geöffnet.

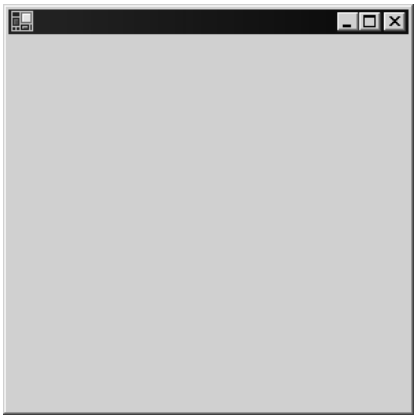

Bild 7.1: Eine einfache Windows-Anwendung ohne Steuerelemente

Das Formular reagiert bereits auf einige Ereignisse, wie z. B. das Schließen über den Schließen-Schalter in der Titelleiste, besitzt aber natürlich noch nicht viel Funktionalität. Um Steuerelemente hinzuzufügen, deren Ereignisse auszuwerten und die Steuerelemente und das Formular zu initialisieren, ist weiterer Quellcode notwendig.

Initialisieren eines Formulars

Um ein Formular zu initialisieren, wird normalerweise im Konstruktor des Formulars eine private Methode aufgerufen, die die Initialisierung übernimmt. Visual Studio nennt diese Methode *InitializeComponent*. In dieser Methode werden Eigenschaften des Formulars eingestellt (Größe, Beschriftung etc.), Steuerelemente erzeugt und diese ebenfalls initialisiert. Der Quellcode eines Formulars, das einen Schalter enthält, sieht dann schon etwas komplexer aus. Ich erläutere den Quellcode mit Hilfe von Kommentaren:

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

namespace Nitty_Gritty.Samples
{
   public class StartForm : System.Windows.Forms.Form
   {
```

```
/* Deklaration einer privaten Eigenschaft für
 * den Schalter */
private System.Windows.Forms.Button btnHello;

/* Der Konstruktor */
public StartForm()
{
   /* Aufruf der privaten Methode zum
    * Initialisieren */
   InitializeComponent();
}

/* Private Methode zur Initialisierung */
private void InitializeComponent()
{
   /* Schalter erzeugen und der privaten
    * Eigenschaft zuweisen */
   this.btnHello = new
      System.Windows.Forms.Button();

   /* Mit der SuspendLayout-Methode wird
    * erreicht, dass das Formular die
    * einzelnen Initialisierungen nicht sofort
    * umsetzt, sondern erst dann, wenn die
    * ResumeLayout-Methode aufgerufen wird.
    * Die Initialisierung wird damit erheblich
    * beschleunigt  */
   this.SuspendLayout();

   /* Initialisierung des Schalters */
   this.btnHello.Name = "btnHello";
   this.btnHello.Text = "Klick mich";
   this.btnHello.Location = new
      System.Drawing.Point(10, 100);
   this.btnHello.Size = new
      System.Drawing.Size(100, 35);
   this.btnHello.TabIndex = 0;
```

```
   /* Damit das Click-Ereignis des Schalters
    * ausgewertet werden kann, wird
    * dieses mit einer Ereignis-Behandlungs-
    * Methode verknüpft (btnHello_Click),
    * die innerhalb der Formularklasse
    * deklariert ist */
   this.btnHello.Click += new
      System.EventHandler(this.btnHello_Click);

   /* Initialisierung des Formulars */
   this.AutoScaleBaseSize = new
      System.Drawing.Size(5, 13);
   this.ClientSize = new
      System.Drawing.Size(300, 150);
   this.Name = "StartForm";
   this.Text = "Hello World";

   /* Hinzufügen des erzeugten Schalters zu den
    * Steuerelementen des Formulars */
   this.Controls.AddRange(new
      System.Windows.Forms.Control[]
      {this.btnHello});

   /* Die Layout-Änderungen übernehmen */
   this.ResumeLayout(false);
}

[STAThread]
static void Main()
{
   /* Starten der Anwendung mit einer neuen
    * Instanz des Formulars */
   Application.Run(new StartForm());
}
/* Ereignis-Behandlungs-Methode für das
 * Click-Ereignis des Schalters */
private void btnHello_Click(object sender,
   System.EventArgs e)
```

```
        {
            /* Ausgabe einer Meldung mit Hilfe der
             * MessageBox-Klasse */
            MessageBox.Show("Hello World");
        }
    }
}
```

Das Ganze sieht ziemlich kompliziert aus. Wenn Sie das Prinzip aber einmal verstanden haben, bleibt nur noch die Arbeit beim Schreiben des Quellcodes übrig. Wenn Sie aber Visual Studio zur Erzeugung von Windows-Anwendungen verwenden, reduziert sich diese Arbeit auf ein Minimum. Visual Studio übernimmt die komplexe Erzeugung der Steuerelemente, der Ereignisprozeduren und die Initialisierung automatisch. Den Quellcode oben habe ich z. B. nicht selbst geschrieben, sondern mit Visual Studio erzeugt (und ein wenig vereinfacht).

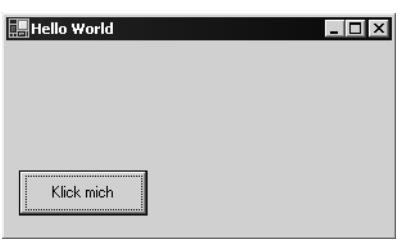

Bild 7.2: Eine einfache Windows-Anwendung

Ereignisbehandlungsmethoden

Um die Ereignisse von Steuerelementen oder Formularen behandeln zu können, benötigen Sie Ereignisbehandlungsmethoden, die (wie Sie ja bereits in Kapitel 4 erfahren haben) immer denselben Grundaufbau besitzen:

```
private void Name(object sender, System.EventArgs e)
{

}
```

Die Argumente der Methode sind bei allen Ereignissen nahezu identisch. Im ersten Argument übergibt das Programm eine Referenz auf das Objekt, das diese Methode aufgerufen hat. Dieses Argument ist dann hilfreich, wenn eine einzige Methode für die Ereignisse verschiedener Objekte verwendet wird. Im zweiten Argument übergibt das Objekt, das das Ereignis ausgelöst hat, weitere Informationen zum Ereignis. Bei einem Tastaturereignis (das bei einer Tastenbetätigung aufgerufen wird) wird hier z. B. die betätigte Taste übergeben. Der Datentyp dieses Arguments ist je nach Ereignis manchmal auch ein anderer als `EventArgs`. Bei Tastaturereignissen wird z. B. der Datentyp `KeyPressEventArgs` übergeben.

Die Ereignisbehandlungsmethode für das `Click`-Ereignis eines Buttons sieht dann z. B. so aus:

```
private void btnRechnen_Click(object sender,
   System.EventArgs e)
{

}
```

Der Name der Methode ist nicht maßgeblich, Sie können jeden beliebigen (gültigen) Namen verwenden. Visual Studio nennt automatisch erzeugte Ereignisbehandlungsmethoden immer nach dem Schema *Objektname_Ereignisname*. Ich halte mich einfach an dieses Schema.

Um die Methode nun mit einem Ereignis zu verbinden, wird dieses in der Initialisierungsmethode der passenden Ereigniseigenschaft des Objekts hinzugefügt. Das folgende Beispiel zeigt einen Ausschnitt aus der Initialisierungsmethode mit dem Quellcode, der den Button erzeugt und initialisiert.

```
private void InitializeComponent()
{
   /* Button erzeugen */
   this.btnRechnen = new System.Windows.Forms.Button();
   /* Erzeugen anderer Steuerelemente */
   ...
   this.SuspendLayout();
```

```
   /* Einstellen der Eigenschaften des Buttons /
   this.btnRechnen.Location = new
      System.Drawing.Point(20, 100);
   this.btnRechnen.Name = "btnDemo";
   this.btnRechnen.TabIndex = 0;
   this.btnRechnen.Text = "Rechnen";

   /* Zuweisen der Ereignisbehandlungsmethode */
   this.btnRechnen.Click += new
      System.EventHandler(this.btnRechnen_Click);
...
```

7.0.2 Der Quellcode eines mit Visual Studio erzeugten Formulars

Der Quellcode eines mit Visual Studio erzeugten Formulars sieht prinzipiell so aus, wie der einer einfachen, selbst entwickelten Windows-Anwendung. Visual Studio fügt der Formularklasse aber noch ein wenig Quellcode hinzu. Die Methode zur Initialisierung (*InitializeComponent*) ist nun in eine Region eingefügt (Abbildung 7.3).

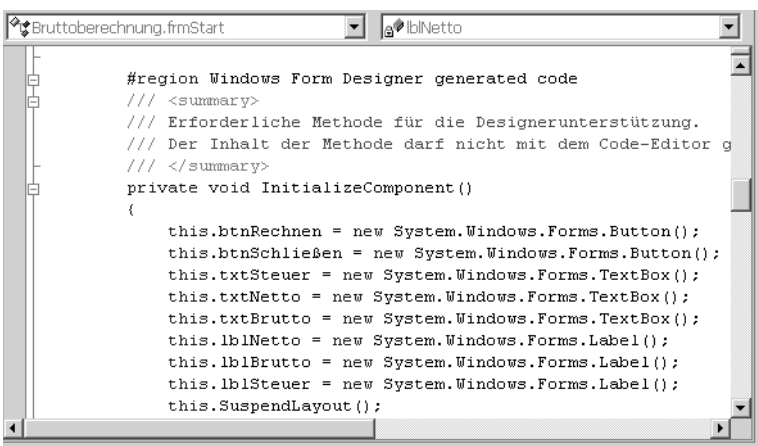

Bild 7.3: Die Initialisierungsmethode einer Windows-Anwendung in Visual Studio

Der Hinweis innerhalb der Region in Abbildung 7.3, der aussagt, dass Sie den Inhalt der Initialisierungsmethode nicht ändern dürfen, ist in meinen Augen so nicht korrekt. Sie können die Initialisierung natürlich auch im Quellcode anpassen. So können Sie z. B. recht einfach die Werte von Eigenschaften ändern. Änderungen im Quelltext sind sofort auch im Designer sichtbar. Achten Sie aber darauf, dass Sie das Erzeugen der Steuerelemente möglichst nicht oder nur mit Bedacht verändern.

Eine weitere Änderung des Quellcodes ist, dass Visual Studio der Formular-Klasse eine private Eigenschaft *components* vom Typ `System.ComponentModel.Container` und eine `Dispose`-Methode hinzugefügt hat. Die Eigenschaft *components* wird in einem einfachen Formular gar nicht verwendet. Erst wenn Sie bestimmte Komponenten, wie z. B. eine `ImageList`-Komponente auf das Formular ziehen, erzeugt Visual Studio in *InitializeComponent* ein Objekt der `System.ComponentModel.Container`-Klasse und gibt dieses bei der Erzeugung der speziellen Komponenten als Container (Behälter) an. Die `Dispose`-Methode zerstört dieses Container-Objekt dann wieder. Diese Methode ersetzt die geerbte `Dispose`-Methode (die aber am Ende aufgerufen wird). `Dispose` wird üblicherweise aufgerufen, wenn ein Objekt zerstört werden soll.

7.1 Application-Objekt und MessageBox

Bei Windows-Anwendungen werden die `Application`- und die `MessageBox`-Klasse sehr häufig verwendet. Ich beschreibe deshalb hier kurz worum es sich dabei handelt.

7.1.1 Die Application-Klasse

Die `Application`-Klasse stellt einige statische Eigenschaften, Methoden und Ereignisse zur Verfügung, über die Sie die Anwendung managen können und Informationen zur Anwendung erhalten.

Element	Beschreibung
`ApplicationExit`	Dieses Ereignis wird aufgerufen, kurz bevor die Anwendung beendet wird. Hier können Sie Ressourcen freigeben, die sonst nirgendwo freigegeben werden. Dateien, die Sie beispielsweise in der `Main`-Methode öffnen, sollten Sie im `ApplicationExit`-Ereignis wieder schließen.
`CurrentCulture`	Diese Eigenschaft bestimmt die aktuelle Kultur, die für die Anwendung verwendet wird (vgl. Kapitel 5).
`ExecutablePath`	Diese Eigenschaft gibt den kompletten Dateinamen der Anwendung zurück.
`ProductName`, `CompanyName`, `ProductVersion`	Diese (schreibgeschützten) Eigenschaften geben den Produktnamen, den Herstellernamen und die Produktversion der Anwendung zurück. Sie definieren diese Werte über die Attribute der Datei *AssemblyInfo.cs*. Die Werte dieser Eigenschaften zeigt Windows in den Eigenschaften der erzeugten *exe*-Datei an.
`StartupPath`	Diese Eigenschaft gibt den kompletten Pfadnamen des Ordners zurück, in dem die Anwendung gespeichert ist.
`void DoEvents()`	Wenn Sie prozessorintensive Aktionen ausführen, kann es sein, dass Windows während der Ausführung der Aktion deutlich blockiert ist. Mit `DoEvents` können Sie innerhalb der Aktion etwas Prozessorzeit abgeben, sodass Windows seine Nachrichtenwarteschlange abarbeiten kann und wieder reaktionsfähig wird.
`void Exit()`	Über diese Methode können Sie die Anwendung beenden. Beachten Sie aber, dass das Beenden eher einem harten Abbruch entspricht als einem sauberen Schließen der Anwendung. `Exit` führt dazu, dass die wichtigen `Closing`-Ereignisse der Formulare nicht aufgerufen werden, in denen Sie u. U. Aufräumarbeiten vornehmen. Beenden Sie eine Anwendung besser, indem Sie das Startformular über dessen `Close`-Methode schließen.

Tabelle 7.1: Die wichtigsten Elemente der Application-Klasse

DoEvents soll ein Beispiel verdeutlichen. Wenn Sie in einer Schleife beispielsweise ein Label aktualisieren wollen,

```
for (int i=0; i<100000; i++)
{
   lblNumber.Text = i.ToString();
}
```

ist Windows durch die prozessorintensive Schleife so stark blockiert, dass zum einen das Label erst am Ende der Schleife aktualisiert wird und zum anderen Windows kaum noch auf Benutzereingaben reagiert (trotz preemptivem Multitasking!).

Wenn Sie innerhalb der Schleife DoEvents aufrufen, geben Sie so viel Prozessorzeit ab, dass Windows wieder arbeiten kann:

```
for (int i=0; i<100000; i++)
{
   lblNumber.Text = i.ToString();
   Application.DoEvents();
}
```

Das Label wird nun aktualisiert und Windows reagiert auf Benutzereingaben. Die Schleife wird aber auch deutlich langsamer ausgeführt. Das sollten Sie immer im Auge behalten.

7.1.2 Die MessageBox-Klasse

Mit der MessageBox-Klasse geben Sie über deren Show-Methode Meldungen aus und erzwingen Entscheidungen vom Anwender. Der Anwender muss einen der angebotenen Schalter betätigen, um den Dialog zu schließen. Die Show-Methode ist mehrfach überladen. Die Syntax kann folgendermaßen dargestellt werden:

```
DialogResult Show(string text
  [, string caption]
    [, MessageBoxButtons buttons]
    [, MessageBoxIcons icon]
    [, MessageBoxDefaultButton defaultButton]
    [, MessageBoxOptions options])
```

```
DialogResult Show(IWin32Window owner, string text
  [, string caption]
    [, MessageBoxButtons buttons]
    [, MessageBoxIcons icon]
    [, MessageBoxDefaultButton defaultButton]
    [, MessageBoxOptions options])
```

Im Argument *text* geben Sie die eigentliche Meldung an. *caption* definiert die Beschriftung des Dialogs. Im Argument *buttons* geben Sie die Schaltflächen an, die die Messagebox besitzen soll. Sie können hier eine der Konstanten der Auflistung `MessageBoxButtons` verwenden: `OK`, `OKCancel`, `AbortRetryIgnore`, `YesNoCancel`, `YesNo` oder `RetryCancel`. Wenn Sie *buttons* nicht angeben, besitzt die Messagebox einen OK-Schalter.

Eine Messagebox, die den Text »Speichern« ausgibt, den Produktnamen der Anwendung als Titel ausgibt und einen *Ja*- und einen *Nein*-Schalter besitzt, öffnen Sie beispielsweise so:

```
if (MessageBox.Show("Speichern?",
   Application.ProductName,
   MessageBoxButtons.YesNo) == DialogResult.Yes)
{
   /* Speichern */
   ...
}
```

Bild 7.4: Eine Messagebox mit buttons = MessageBoxButtons.YesNo

Die `Show`-Methode gibt einen Wert vom Typ `DialogResult` zurück, der den betätigten Schalter definiert. Die möglichen Werte sind `OK`, `Cancel`, `Abort`, `Retry`, `Ignore`, `Yes` und `No`.

Im Argument *icon* können Sie definieren, welches Icon der Dialog anzeigt. Der Datentyp dieses Arguments (`MessageBoxIcon`) ist eine Auflistung mit den folgenden Werten:

- `None`: Kein Icon.
- `Hand`, `Error`, `Stop`: Weißes X in einem roten Kreis. Für Fehlermeldungen.
- `Question`: Fragezeichen.
- `Exclamation`, `Warning`: Ausrufezeichen für Warnungen.
- `Asterisk`, `Information`: Informationszeichen für informelle Meldungen.

Fragen Sie mich nicht, warum für dasselbe Icon teilweise verschiedene Konstanten definiert sind ...

Das Argument *defaultButton* bestimmt, welcher Schalter derjenige ist, der mit ↵ betätigt werden kann, wenn die Messagebox mehrere Schalter besitzt.

Das folgende Beispiel fragt den Anwender im `Closing`-Ereignis eines Formulars, ob er wirklich beenden will. Als Icon wird das Fragezeichen eingestellt, der Defaultbutton wird auf den Nein-Schalter gesetzt:

```
private void StartForm_Closing(object sender, System.ComponentMo-
del.CancelEventArgs e)
{
   if (MessageBox.Show("Wirklich beenden?",
      Application.ProductName,
      MessageBoxButtons.YesNo,
      MessageBoxIcon.Question,
      MessageBoxDefaultButton.Button2)
      == DialogResult.No)
   {
      /* Das Beenden abbrechen */
      e.Cancel = true;
   }
}
```

Bild 7.5: Eine Messagebox mit Icon, zwei Schaltern und Default-Button auf dem zweiten Schalter

Die Messagebox können Sie auch mit einem Owner-Formular öffnen, das Sie im ersten Argument angeben. Der Dialog wird dann im Vordergrund vor dem Owner-Formular angezeigt. Normalerweise wird die Messagebox vor dem Formular angezeigt, von dem aus sie geöffnet wurde. Den Sinn, ein anderes Owner-Formular anzugeben, habe ich nie verstanden ...

Im letzten Argument können Sie schließlich noch einige Optionen übergeben, die nur selten benötigt werden (und deshalb hier nicht besprochen werden).

7.2 Steuerelemente und Komponenten

Auf den folgenden Seiten beschreibe ich den Umgang mit den wichtigsten Steuerelementen und Komponenten. Ich gehe davon aus, dass Sie Visual Studio verwenden und die Steuerelemente bzw. Komponenten folglich nicht »von Hand« (im Quellcode) erzeugen müssen. Aufgrund der Vielfalt der Steuerelemente kann ich leider nur die wichtigsten (die, die in der Praxis am häufigsten verwendet werden) beschreiben, und davon auch nur die gängigen Eigenschaften, Methoden und Ereignisse.

7.2.1 Grundlagen

Meine Namenskonvention

Für Steuerelemente und Komponenten verwende ich eine Namenskonvention. Der Name beginnt immer mit einem möglichst dreistelligen, kleingeschriebenen Präfix, den ich aus dem Klassennamen des Steuerelements ableite. »txt« steht beispielsweise für »TextBox«. Diese Konvention weicht zwar von den allgemeinen C#-Namensrichtlinien ab, besitzt aber einige Vorteile:

- Ich erkenne im Quellcode, dass es sich um ein Steuerelement handelt.
- Wenn ich IntelliSense nutze, finde ich ein Steuerelement oder eine Komponente im Quelltext sehr schnell. Wenn ich beispielsweise eine Textbox ansprechen will, deren Name mir entfallen ist (was bei mir öfter vorkommt) schreibe ich einfach »txt« und betätige

dann Strg Leertaste. Die IntelliSense-Liste zeigt dann alle verfügbaren Elemente an und hat die Markierung bereits auf die erste Textbox gestellt.

Ich erläutere die einzelnen Präfixe bei der Beschreibung der Steuerelemente.

Steuerelemente und Komponenten

Steuerelemente sind Klassen, die eine Oberfläche besitzen, die im Entwurf und in der Laufzeit sichtbar ist. Steuerelemente werden automatisch instanziert, wenn sie auf einen Container (z. B. ein Formular) gezogen werden. Komponenten sind Steuerelementen ähnlich. Der einzige Unterschied ist, dass diese keine sichtbare Oberfläche besitzen, wenn das Programm ausgeführt wird. Die Eigenschaften einer Komponente können Sie, genau wie bei einem Steuerelement, über das Eigenschaftenfenster bearbeiten und deren Ereignisse auswerten. Die automatische Instanzierung, die im Eigenschaftenfenster dargestellten Eigenschaften und die automatische Erzeugung von Ereignisprozeduren sind die Vorteile einer Komponente gegenüber einer normalen Klasse. Komponenten werden automatisch in einem speziellen Bereich unterhalb des Formulars angeordnet.

Positions- und Größenangaben

Positions- und Größenangaben sind in der `Windows.Forms`-Bibliothek Strukturen. Positionsangaben werden in einer Instanz der Struktur `System.Drawing.Point` angegeben. Die wichtigen Eigenschaften sind `x` und `y`. Im Konstruktor können Sie diese Werte direkt übergeben. Die Größe wird über eine Instanz der Struktur `System.Drawing.Size` angeben. Die wichtigen Eigenschaften sind `Width` und `Height`, die Sie ebenfalls direkt im Konstruktor übergeben können. Die Konstruktoren beider Strukturen erlauben als Argument auch jeweils eine Instanz der anderen Struktur.

Im Quellcode (nicht im Eigenschaftenfenster) können Sie aber auch auf die Eigenschaften `Left`, `Top`, `Width` und `Height` zugreifen, um die Position und Größe abzufragen oder zu setzen.

Die Einheit für Positions- und Größenangaben ist Pixel.

Farbwerte

Eigenschaften, die Farbwerte speichern, besitzen den Typ `System.Drawing.Color`. `Color` ist eine Klasse, die konstante Eigenschaften für die gängigen Farben besitzt. Wenn Sie eine Farbe im Programm ändern wollen, können Sie direkt eine der Farbeigenschaften zuweisen:

```
txtFirstName.BackColor = Color.Red;
```

Ich verzichte im Beispiel auf die Angabe des Namensraums `System.Drawing`, weil dieser üblicherweise in einer `using`-Anweisung angegeben ist.

Sie können aber natürlich auch einen beliebigen Farbwert verwenden. Diesen können Sie dann als RGB-Wert über die Methode `FromArgb` angeben:

```
txtFirstName.BackColor = Color.FromArgb(255,128,0);
```

Im ersten Argument definieren Sie den Rotanteil, im zweiten den Grünanteil und im dritten den Blauanteil mit Werten zwischen 0 und 255. Das Beispiel erzeugt ein (schönes) Orange. Alternativ können Sie den Farbwert auch als `int`-Wert übergeben (ein RGB-Wert ist eigentlich ein `int`-Wert, der die drei Farben in den ersten drei Byte definiert).

Über die Methode `FromName` können Sie einen Farbwert auch als String definieren:

```
txtFirstName.BackColor = Color.FromName("Orange");
```

Einige Farbwerte sind Systemfarben. Windows definiert ja für alle Windowselemente eigene Farben, die der Anwender einstellen kann. Die Steuerelemente sind per Voreinstellung auf diese Systemfarben eingestellt. Im Programm können Sie die Systemfarben über die Auflistung `System.Drawing.KnownColor` einstellen, die alle Systemfarbwerte enthält. Einen solchen Farbwert müssen Sie allerdings der `FromKnownColor`-Methode der `Color`-Klasse übergeben, um einen gültigen `Color`-Farbwert zu erhalten.

Wenn Sie verschiedenen Farbwerte, wie z. B. einen 32-Bit-Windows-Farbwert oder eine HTML-Farbangabe, als Farbe verwenden wollen, können Sie die Methoden der Klasse `System.Drawing.ColorTranslator` zur Übersetzung dieser Farbwerte verwenden.

Die Schriftart

Die Schriftart wird über ein Objekt der Klasse `Font` definiert. Diese Klasse besitzt die wichtigen Eigenschaften `Bold` (fett), `Italic` (kursiv), `Name` (der Name der Schrift), `Overline` (überstrichen), `Size` (Größe), `Strikeout` (durchgestrichen) und `Underline` (unterstrichen).

Ereignisse als behandelt definieren

Die Klasse `EventArgs`, von der alle Ereignisargumentklassen erben, besitzt die Eigenschaft `Handled`. Wenn Sie diese Eigenschaft auf `true` setzen, definieren Sie das Ereignis als behandelt. Das Steuerelement oder die Komponente wertet eventuelle Eingaben dann nicht mehr aus. Ein Beispiel dafür finden Sie im nächsten Abschnitt.

Auswerten von Tastaturereignissen

Alle Steuerelemente, die eine Eingabe erlauben, ermöglichen auch die Auswertung von Tastaturereignissen. Dazu stehen Ihnen die Ereignisse `KeyDown`, `KeyPress` und `KeyUp` zur Verfügung. `KeyDown` und `KeyUp` werden für alle Tasten aufgerufen. Im zweiten Argument der Ereignisbehandlungsmethode können Sie über die Eigenschaft `KeyCode` erfahren, welche Taste betätigt wurde. Vergleichen Sie den Wert dieser Eigenschaft mit einer der in der Aufzählung `System.Windows.Forms.Keys` gespeicherten Konstanten. Die Konstante `F1` steht beispielsweise für die F1-Taste. Über die Eigenschaften Alt, Control und Shift können Sie herausfinden, ob Alt, Strg und/oder ⇧ gleichzeitig mit einer anderen Taste oder auch allein betätigt wurden. Das folgende Beispiel eignet sich als Fingerübung zur Vermeidung von Muskelverspannungen in der linken Hand:

```
private void txtDemo_KeyDown(object sender,
   System.Windows.Forms.KeyEventArgs e)
{
   if (e.Shift && e.Control && e.Alt &&
     (e.KeyCode == System.Windows.Forms.Keys.F1))
      /* Shift+Strg+Alt+F1 */
      MessageBox.Show("Wow, was für ein Griff.");
}
```

Das Ereignis KeyPress wird nur dann aufgerufen, wenn der Anwender eine der so genannten ASCII-Tasten vollständig betätigt. Zu den ASCII-Tasten gehören eigentlich nur die Tasten im Hauptblock der Tastatur, mit Ausnahme der Tasten ⇧, Strg, Alt, AltGr und den Windowstasten. Als Ausnahme gehört die Esc-Taste auch noch zu den ASCII-Tasten. Die ASCII-Tasten besitzen einen ASCII-Code und korrespondieren deswegen mit einem Zeichen. Die Return-Taste besitzt z. B. den Code 13, der für das Zeichen '\r' steht.

Um nun eine Betätigung einer ASCII-Taste abzufangen, werden Sie die Eigenschaft KeyChar des zweiten Arguments der Ereignisbehandlungsmethode aus. Hier können Sie mit dem Zeichen oder direkt mit dem ASCII-Wert vergleichen:

```
private void txtDemo_KeyPress(object sender, System.Windows.Forms.Key-
PressEventArgs e)
{
   switch (e.KeyChar)
   {
      case 13: // oder '\r'
         MessageBox.Show("Return");
         /* Piepsen verhindern */
         e.Handled = true;
         break;
      case 27:
         MessageBox.Show("Esc");
         break;
      case 8:
         MessageBox.Show("Backspace");
         break;
   }
}
```

Das Beispiel enthält einen wichtigen Trick: Wenn Sie das Ereignis als behandelt definieren, wird die Tastenbetätigung nicht an die Textbox selbst weitergegeben. Im Fall von Return verhindert dies das voreingestellte nervige Piepsen.

Container- und Kind-Steuerelemente

Einige Steuerelemente können andere in sich aufnehmen. Sie dienen dann als Container für so genannte Kind-Steuerelemente. Das Panel ist ein Beispiel dafür. Ziehen Sie andere Steuerelemente einfach auf das Container-Steuerelement, um diese darauf zu platzieren. Wenn ein Steuerelement direkt auf einem Formular angelegt wurde, ist das Formular der Container.

Bilder in Steuerelementen

Einige Steuerelemente können Bilder in den Formaten Bitmap, JPEG, GIF, PNG, Icon, EMF und WMF anzeigen. Normalerweise weisen Sie diese Bilder im Entwurf über das Eigenschaftenfenster zu. Visual Studio speichert diese Bilder dann in einer Ressourcendatei, die mit dem jeweiligen Formular verbunden ist und die denselben Namen mit der Endung *resx* erhält. In *InitializeComponent* erzeugt Visual Studio eine Instanz der Klasse `RessourcenManager` für das Formular. Mit Hilfe dieses Managers werden die Bilder dann aus der Ressource ausgelesen.

Sie können Bilder aber auch dynamisch in der Laufzeit einlesen. Dazu benötigen Sie einen `FileStream`, der die Bilddatei repräsentiert. Diesen Stream übergeben Sie dann einer neuen Instanz der Klasse `Bitmap`. Das folgende Beispiel lädt ein Bild in ein `PictureBox`-Steuerelement:

```
string filename = "C:\Winnt\Kaffeetasse.bmp";
System.IO.FileStream f = new System.IO.FileStream(
   filename,System.IO.FileMode.Open);
pictureBox.Image = new System.Drawing.Bitmap(f);
```

7.2.2 Wo initialisieren?

Häufig müssen Sie Steuerelemente oder Komponenten beim Öffnen eines Formulars mit dynamisch im Programm ermittelten Werten initialisieren. Eine wichtige Frage dabei ist, an welcher Stelle im Quellcode Sie diese Initialisierung vornehmen. Wenn Sie Visual Studio nicht verwenden, ist die Antwort einfach: Im Konstruktor des Formulars, da wo Sie die Steuerelemente auch erzeugen. Wenn Sie aber Visual Studio einsetzen, ist die Frage schon nicht mehr so einfach zu beantworten. Visual Studio verwendet ja die Methode *InitializeCom-*

ponent, in der die Steuerelemente erzeugt werden und die Werte, die Sie im Eigenschaftenfenster gesetzt haben, in die Eigenschaften der Steuerelemente geschrieben werden. Visual Studio fügt im Konstruktor des Formulars eine Hinweis ein, dass Sie Initialisierungscode erst nach dem Aufruf von *InitializeComponent* einfügen sollen/dürfen. Wenn Sie nun aber nach dem Aufruf von *InitializeComponent* beispielsweise einem Label eine Beschriftung geben, ist die Initialisierung des Label eigentlich doppelt vorhanden: Einmal in *InitializeComponent* und einmal danach. Wenn Sie nun daran denken, die Initialisierung gleich in *InitializeComponent* vorzunehmen, müssen Sie ein wenig aufpassen. Visual Studio macht absolut keine Probleme, wenn Sie konstante Werte in Eigenschaften der Steuerelemente schreiben. Nur wenn Sie den Wert einer Eigenschaft dynamisch über einen Ausdruck (z. B. einen Methodenaufruf) ermitteln, kann Visual Studio damit nicht korrekt umgehen: Das Eigenschaftenfenster zeigt zwar zunächst – wenn möglich – das Ergebnis des Ausdrucks an. Wenn Sie dann aber nur eine Eigenschaft des Formulars oder eines seiner Steuerelemente über das Eigenschaftenfenster ändern, überschreibt Visual Studio die Initialisierung mit dem aktuellen Ergebnis des Ausdrucks. Ein Beispiel soll dies verdeutlichen: Platzieren Sie ein Label auf einem Formular und nennen Sie dieses *lblDate*. Das Label soll beim Instanzieren des Formulars das aktuelle Datum anzeigen. Ändern Sie die Initialisierung in *InitializeComponent* folgendermaßen ab:

```
this.lblDate.Location =
   new System.Drawing.Point(8, 16);
this.lblDate.Name = "lblDate";
this.lblDate.Size = new System.Drawing.Size(280, 24);
this.lblDate.TabIndex = 0;
this.lblDate.Text =
   System.DateTime.Now.ToShortDateString();
```

Wenn Sie das Programm nun ausführen, funktioniert das soweit. Nun ändern Sie über das Eigenschaftenfenster eine beliebige Eigenschaft und schauen sich danach die Initialisierung in der Formular-Klasse an. Visual Studio hat das aktuelle Datum als Stringkonstante eingefügt:

```
this.lblDate.Text = "25.11.2001"
```

Nehmen Sie also alle dynamischen Initialisierungen prinzipiell erst nach dem Aufruf von *InitializeComponent* vor. Einige dynamische Initialisierungen sind davon aber wieder ausgeschlossen: Für Eigenschaften, die Referenztypen sind, erzeugt auch Visual Studio die Instanzen in *InitializeComponent*. Alle statischen Initialisierungen sollten Sie besser in *InitializeComponent* implementieren, damit Ihr Programmcode möglichst performant bleibt.

Die Klasse Form besitzt übrigens noch das `Load`-Ereignis. Dieses Ereignis wird aufgerufen, kurz bevor das Formular angezeigt wird. In diesem Ereignis können Sie ebenfalls initialisieren. Sie können hier auf alle Steuerelemente zugreifen, weil diese zum Zeitpunkt des `Load`-Ereignisses bereits erzeugt und (vor-)initialisiert sind. Formulare werden ab Seite 352 behandelt.

7.2.3 Automatische Positionierung und Größenanpassung

Die Steuerelemente der `Windows.Forms`-Bibliothek besitzen ein nettes Feature: Sie sind in der Lage, sich auf einem Container automatisch auszurichten und/oder ihre Größe anzupassen, wenn der Container in der Größe verändert wird. Wenn Sie zulassen, dass der Anwender ein Formular vergrößern und verkleinern kann, können Sie so sehr einfach sicherstellen, dass die Steuerelemente passend zur Formulargröße angeordnet und/oder in der Größe angepasst werden. Alle Steuerelemente besitzen dazu die Eigenschaften `Dock` und `Anchor`.

Wenn Sie diese Eigenschaften mit einer TextBox ausprobieren, beachten Sie, dass Sie die Eigenschaft `MultiLine` *der TextBox auf* `true` *setzen müssen, damit diese in der Höhe dynamisch verändert werden kann.*

Über die Eigenschaft `Dock` können Sie bestimmen, an welchen Seiten des Containers sich ein Steuerelement andockt. `Dock` ist eine Auflistung vom Typ `DockStyle` mit den Werten `Top`, `Left`, `Right`, `Bottom`, `All` (Andocken an allen Seiten) und `None` (kein Andocken). Das Steuerelement wird dabei automatisch in der Größe bzw. Position angepasst, wenn der Container in der Größe verändert wird. Container besitzen die Eigenschaft `DockPadding`, über die Sie den Rand zwischen dem

Container und dem Steuerelement für alle vier Seiten getrennt bestimmen können.

Über die Eigenschaft `Anchor` können Sie den »Anker« bestimmen, an dem ein Steuerelement sich am Container festsetzt. `Anchor` setzt sich aus den Werten der Aufzählung `AnchorStyles` zusammen: `Top`, `Left`, `Right`, `Bottom` und `None`. Ein Anker bewirkt, dass der Platz zwischen Steuerelement und Container für diesen Anker immer gleich bleibt. Das Steuerelement wird bei einem Resize des Containers je nach den gesetzten Ankerpunkten verschoben oder vergrößert. Wenn Sie beispielsweise Anker für `Top` und `Left` (Voreinstellung) setzen, bleibt das Steuerelement immer an seiner linken, oberen Position und wird nicht in der Größe angepasst. Setzen Sie zusätzlich rechts einen Anker, wird das Steuerelement bei einem Resize des Containers in der Größe nach rechts so angepasst, dass der Zwischenraum zwischen dem rechten Rand des Steuerelements und dem rechten Rand des Containers immer gleich bleibt. Setzen Sie nur Anker für oben und rechts, wird das Steuerelement bei einem Resize nach rechts bzw. links verschoben. Wenn Sie übrigens keinen Anker definieren und das Steuerelement im Entwurf in der Mitte des Containers platzieren, bleibt es immer in der Mitte. Probieren Sie das einfach einmal aus. Diese Eigenschaft funktioniert bereits, wenn Sie das Formular im Formulardesigner in der Größe verändern.

Um zu verhindern, dass der Anwender ein Formular zu sehr verkleinert, sodass Steuerelemente nicht mehr sichtbar sind, können Sie in der Eigenschaft `MinimumSize` *des Formulars eine Minimalgröße angeben. Diese können Sie recht einfach über ein Verkleinern des Formulars im Entwurf ermitteln.*

7.2.4 Die Basisklasse Control

Alle Steuerelemente sind von der Klasse `Control` abgeleitet und besitzen deswegen die Eigenschaften, Methoden und Ereignisse dieser Klasse. Die wichtigsten liste ich in Tabelle 7.2, Tabelle 7.3 und Tabelle 7.4 auf.

Alle Steuerelemente sind zwar von `Control` *abgeleitet und besitzen deshalb prinzipiell alle Elemente dieser Klasse. Bei Steuerelementen, bei denen die Anwendung eines Elements aber keinen Sinn macht, wird dieses in der Steuerelementklasse häufig komplett ausgeblendet (vgl. Kapitel 4). Die Eigenschaft* `Text` *werden Sie beispielsweise bei einem* `PictureBox`*-Steuerelement vergeblich suchen. Andere, für das jeweilige Steuerelement sinnlose Elemente sind zwar vorhanden, werden aber nicht im Eigenschaftenfenster angezeigt. Die Tastaturereignisse eines Labels, die nicht im Eigenschaftenfenster aufgelistet werden, aber dennoch vorhanden sind, sind Beispiele dafür.*

Die wichtigsten Eigenschaften

Eigenschaft	Beschreibung
`BackColor`	definiert die Hintergrundfarbe
`BackgroundImage`	hier können Sie bei einigen Steuerelementen eine Bilddatei angeben, die als Hintergrund verwendet wird. Bei einigen Steuerelementen, wie z. B. der TextBox, ist diese Eigenschaft verborgen.
`CausesValidation`	legt fest, ob ein Fokuswechsel auf dieses Steuerelement eine Validierung des Steuerelements bewirkt, das den Fokus zuvor besaß. Validierungen werden ab Seite behandelt.
`ContainsFocus`	Wenn diese Eigenschaft `true` ist, besitzt das Steuerelement oder eines seiner Kind-Elemente gerade den Fokus.
`ContextMenu`	Dieser Eigenschaft weisen Sie ein `ContextMenu`-Steuerelement zu, das das Kontextmenü des Steuerelements definiert.
`Cursor`	definiert die Art des Mauscursors, wenn dieser sich über dem Steuerelement befindet
`Enabled`	Über diese boolesche Eigenschaft können Sie das Steuerelement deaktivieren, sodass der Anwender damit nicht mehr arbeiten kann.

Eigenschaft	Beschreibung
`Focused`	gibt `true` zurück, wenn das Steuerelement gerade den Eingabefokus besitzt
`Font`	definiert die Schriftart
`ForeColor`	definiert die Farbe
`Height`	definiert die Höhe
`Left`	definiert die linke Position relativ zum Container
`Name`	definiert den Namen des Steuerelements
`TabIndex`	definiert, auf welcher Position dieses Steuerelement in der Tabulatorreihenfolge liegt
`TabStop`	legt fest, ob das Steuerelement mit der Tab-Taste angesprungen werden kann
`Tag`	In dieser Eigenschaft können Sie beliebige eigene Daten ablegen, die Sie im Programm wieder auslesen können.
`Text`	definiert die Beschriftung oder einen textuellen Inhalt des Steuerelements
`Top`	definiert die obere Position relativ zum Container
`Visible`	Über diese Eigenschaft können Sie ein Steuerelement unsichtbar und wieder sichtbar schalten.
`Width`	definiert die Breite

Tabelle 7.2: Die wichtigsten Eigenschaften der Basisklasse Control

Die wichtigsten Methoden

Methode	Beschreibung
`void BringToFront()`	holt ein Steuerelement, das gerade von anderen Steuerelementen (teilweise) verdeckt wird, nach vorne
`bool Focus()`	setzt den Eingabefokus auf das Steuerelement. Gibt `true` zurück, wenn das Steuerelement erfolgreich den Fokus erhalten hat (in einigen Situationen ist das nicht möglich, z. B. dann, wenn das Steuerelement deaktiviert oder unsichtbar ist).
`void ResetCursor()`	setzt den Cursor wieder auf den Default-Cursor zurück (nachdem Sie diesen über die `Cursor`-Eigenschaft geändert haben)
`void ResetForeColor()`	setzt die Vordergrundfarbe auf den Defaultwert zurück
`void Scale(float ratio)` `void Scale(float dx, float dy)`	skaliert das Steuerelement und seine Kind-Elemente auf eine andere Größe. Der Wert 0.5F steht z. B. für die halbe Größe, der Wert 2F steht für eine doppelte Größe.
`void SendToBack()`	positioniert das Steuerelement nach hinten, sodass es von anderen Steuerelementen verdeckt werden kann
`void Show()`	zeigt ein zuvor unsichtbares Steuerelement an

Tabelle 7.3: Die wichtigsten Methoden der Basisklasse Control

Die wichtigsten Ereignisse

Tabelle 7.4 stellt die wichtigsten Ereignisse der Basisklasse `Control` dar. Beachten Sie, dass alle Steuerelemente wenigstens einen Teil dieser Ereignisse besitzen (bei anderen sind einige Ereignisse, die prinzipiell nicht zum Steuerelement passen, ausgeblendet). Ein Label, das eigentlich nur zur Ausgaben von Text und Grafik verwendet wird, kann beispielsweise auf die Ereignisse `Click`, `DoubleClick` und auf die Mausereignisse reagieren.

Ereignis	Beschreibung
`Click`	wird aufgerufen, wenn der Anwender in das Steuerelement mit der Maus klickt. Bei Buttons wird dieses Ereignis auch dann aufgerufen, wenn der Anwender den Button über die Tastatur betätigt.
`DoubleClick`	wird bei einem Doppelklick auf dem Steuerelement aufgerufen
`Enter`	wird aufgerufen, wenn der Anwender in ein Steuerelement wechselt, aber bevor dieses den Eingabefokus erhält
`GotFocus`	wird aufgerufen, wenn ein Steuerelement den Eingabefokus erhält. Dieses Ereignis ist allerdings im Eigenschaftenfenster meist ausgeblendet. Ich denke, es existiert nur aus Kompatibilitätsgründen zu alten COM-Steuerelementen. Verwenden Sie besser `Enter`.
`KeyDown`	wird aufgerufen, wenn der Anwender eine beliebige Taste drückt
`KeyPress`	wird aufgerufen, nachdem der Anwender eine ASCII-Taste betätigt hat. Dieses Ereignis wird nicht für die Sondertasten (F1-F12, Cursor, Pos1, Entf etc.) aufgerufen.
`KeyUp`	wird aufgerufen, wenn der Anwender eine beliebige Taste loslässt
`Leave`	wird aufgerufen, wenn der Anwender ein Steuerelement verlassen will, aber bevor dieses den Eingabefokus verliert. Hier können Sie das Verlassen abbrechen.
`LostFocus`	wird aufgerufen, wenn das Steuerelement den Eingabefokus verloren hat. Wie `GotFocus` ist dieses Ereignis im Eigenschaftenfenster meist nicht sichtbar.
`MouseDown`	wird aufgerufen wenn der Anwender eine Maustaste auf dem Steuerelement gedrückt hat
`MouseEnter`	wird aufgerufen, wenn der Mauszeiger in das Steuerelement fährt

Ereignis	Beschreibung
MouseHover	wird aufgerufen, wenn der Mauszeiger auf das Steuerelement bewegt wird und dort einen kurzen Moment stehenbleibt. Wenn der Anwender den Mauszeiger nur über das Steuerelement zieht, wird dieses Ereignis nicht aufgerufen.
MouseLeave	wird aufgerufen, wenn der Mauszeiger das Steuerelement verlässt
MouseMove	wird aufgerufen, wenn der Mauszeiger auf dem Steuerelement bewegt wird
MouseUp	wird aufgerufen, wenn der Anwender eine Maustaste auf dem Steuerelement loslässt
MouseWheel	wird aufgerufen, wenn das Mausrad betätigt wird, während der Mauszeiger auf dem Steuerelement verweilt
TextChanged	wird aufgerufen, nachdem der Inhalt der Eigenschaft Text durch den Anwender oder durch das Programm geändert wurde
Validated	wird aufgerufen, wenn das Steuerelement eine Validierung abgeschlossen hat (nach Validating)
Validating	Dieses Ereignis wird aufgerufen, nachdem der Anwender das Steuerelement verlassen hat (nach dem Leave-Ereignis). Hier können Sie Validierungen vornehmen und das Verlassen des Steuerelements u. U. verhindern.

Tabelle 7.4: Die wichtigsten Ereignisse der Basisklasse Control

7.2.5 Label

Ein Label ist ein reines Textausgabefeld, das als Bezeichnungsfeld oder als Ausgabefeld genutzt wird. Daneben können Sie einem Label über Drag & Drop auch Text übergeben, was aber wahrscheinlich nicht oft verwendet wird.

Die wichtigste Eigenschaft ist wohl Text, hier setzen Sie die Beschriftung des Labels. Wenn AutoSize false ist, wird der Text automatisch am rechten Rand umbrochen. Wenn Sie die Eigenschaft AutoSize auf true setzen, wird das Label automatisch in der Breite vergrößert, sodass der enthaltene Text komplett in das Label passt. Über die Eigenschaft TextAlign können Sie die Ausrichtung des Textes im Label be-

stimmen. Wenn Sie zusätzlich zum Text ein Bild ausgeben wollen, geben Sie dieses über die Eigenschaft `Image` an. Alternativ können Sie auch ein `ImageList`-Steuerelement (das eine Liste von Bildern verwaltet) in der gleichnamigen Eigenschaft angeben und das anzuzeigende Bild über `ImageIndex` spezifizieren.

Soll das Label als Beschriftung für ein anderes Steuerelement dienen, das den Fokus erhalten kann, können Sie durch das einfache Vorstellen eines »&« vor ein Zeichen in der Beschriftung einen Shortcut erzeugen. Dazu muss allerdings `UseMnemonic true` sein und das Label in der Tabulatorreihenfolge genau eine Position vor dem anderen Steuerelement liegen.

7.2.6 LinkLabel

Das LinkLabel (mein Präfix: *llb*) ist ein erweitertes Label, das wie ein HTML-Link dargestellt wird. Der Anwender kann den Link betätigen. Im `LinkClicked`-Ereignis werten Sie diese Betätigung aus. Die `LinkLabel`-Klasse ist direkt von der `Label`-Klasse abgeleitet, besitzt also alle Elemente dieser Klasse.

Die Eigenschaft `Text` enthält den kompletten dargestellten Text des Labels. In der Eigenschaft `LinkArea` können Sie festlegen, welcher Teil des Textes den Link darstellt. Voreingestellt ist, dass der gesamte Text als Link verwendet wird. Ein LinkLabel, das im Text »In unserem Shop können Sie online einkaufen« nur das Wort »Shop« als Link darstellen soll, wird so initialisiert:

```
this.llbDemo.Text = "In unserem Shop können Sie " +
   "online einkaufen.";
this.llbDemo.LinkArea = new LinkArea(11, 4);
```

Wenn Sie mehrere Bereiche im Text als Link darstellen wollen, müssen Sie diese dynamisch der `Link`-Auflistung des Label hinzufügen:

```
/* LinkLabel initialisieren */
this.llbDemo.Text = "In unserem Shop können Sie " +
   "online einkaufen.\nBeachten Sie auch " +
   "unsere Angebote.";
this.llbDemo.Links.Add(11,4,"Shop");
this.llbDemo.Links.Add(70,8,"Supplies");
```

In den ersten beiden Argumenten der `Add`-Methode übergeben Sie die Position und die Breite des Links im Text, im optionalen letzten Argument können Sie beliebige Daten übergeben, die Sie im `LinkClicked`-Ereignis dann auswerten können. Das '\n' im Text sorgt übrigens für einen Zeilenumbruch.

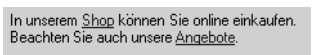

Bild 7.6: Ein LinkLabel mit zwei Links

Um die Betätigung eines Links auszuwerten, fragen Sie im `LinkClicked`-Ereignis die Eigenschaft `LinkData` des `Link`-Objekts, das im letzten Argument als Eigenschaft übergeben wird, ab. Um den Link als »besucht« zu kennzeichnen, können Sie die Eigenschaft `Visited` auf `true` setzen. Der Link erhält dann (normalerweise) automatisch eine andere Farbe:

```
private void llbDemo_LinkClicked(object sender, System.Win-
dows.Forms.LinkLabelLinkClickedEventArgs e)
{
   MessageBox.Show("Link '" + e.Link.LinkData +
     "' des LinkLabel wurde betätigt.");
   e.Link.Visited = true;
}
```

Die Eigenschaften `LinkColor`, `VisitedLinkColor`, `DisabledLinkColor` und `ActiveLinkColor` legen schließlich noch die Farbe der Links fest. Über die Eigenschaft `LinkBehavior` können Sie festlegen, ob und wann der Unterstrich im Link angezeigt wird.

7.2.7 Button

Der Button (mein Präfix: *btn*) ist ein sehr einfaches Steuerelement. Im `Click`-Ereignis werten Sie die Betätigung des Buttons aus. Die Eigenschaft `Text` bestimmt die Beschriftung. In der Eigenschaft `TextAlign` können Sie die Ausrichtung des Textes bestimmen. Soll der Button zusätzlich zum Text ein Bild darstellen, können Sie dieses ich die Eigenschaft `Image` laden. Wie beim Label können Sie stattdessen auch ein `ImageList`-Steuerelement verwenden. Die Eigenschaft `ImageAlign` bestimmt die Ausrichtung des Bildes.

Über die Eigenschaft `DialogResult` bestimmen Sie, welchen Wert der Button in die gleichnamige Eigenschaft des Formulars schreibt, wenn er betätigt wird. Wenn Sie ein Formular über dessen `OpenDialog`-Methode als Dialog öffnen (der geschlossen werden muss, damit das Programm weiterläuft), gibt diese Methode den Wert der `DialogResult`-Eigenschaft zurück und Sie können dadurch ermitteln, welcher Button betätigt wurde.

7.2.8 TextBox

Die Textbox (mein Präfix: *txt*) ist ein Texteingabefeld, in dem einzeiliger und mehrzeiliger Text dargestellt werden kann. Die Textbox unterstützt die Zwischenablage mit den üblichen Tastenkombinationen Strg C, Strg X und Strg V. Natürlich sind die üblichen Editierfunktionen (Löschen, Einfügen etc.) auch bereits integriert.

In der Eigenschaft `Text` wird der Text des Eingabefeldes verwaltet. Unter Windows 95/98/Me können Sie hier nur maximal 65 KB Text speichern. Unter Windows NT/2000/XP kann die TextBox so viel Text speichern, wie der Arbeitsspeicher zulässt. Achten Sie darauf, dass die `MaxLength`-Eigenschaft, die die maximale Eingabelänge bestimmt, eine Voreinstellung besitzt.

Wenn Sie im Programm Text hinzufügen wollen, können Sie dazu die `Text`-Eigenschaft verwenden:

```
txtDemo.Text = "Ich bin";
txtDemo.Text += "eine Textbox";
```

oder die `AppendText`-Methode:

```
txtDemo.Text = "Ich bin";
txtDemo.AppendText("eine Textbox");
```

Über die Eigenschaft `CharacterCasing` können Sie festlegen, dass der eingegebene Text normal, in Groß- oder in Kleinbuchstaben erscheint. Die `MaxLength`-Eigenschaft bestimmt, wie viele Zeichen der Anwender maximal eingeben kann. Über `ReadOnly` können Sie bestimmen, dass der Inhalt der Textbox vom Anwender nur gelesen werden darf. Das Programm kann aber trotzdem Text hineinschreiben.

Passwort-Textboxen

Textboxen können für die Eingabe von Passwörtern verwendet werden. Schreiben Sie dazu ein beliebiges Zeichen (normalerweise *) in die Eigenschaft `PasswordChar`. Alle eingegebenen Zeichen werden dann durch das Passwortzeichen dargestellt. Den eingegebenen Text können Sie natürlich trotzdem ganz normal aus der Eigenschaft `Text` auslesen.

Mehrzeilige Textboxen

Über `MultiLine` können Sie die Textbox so einrichten, dass diese einen mehrzeiligen Text erlaubt. In diesem Fall sind häufig Bildlaufleisten sinnvoll, die Sie über die Eigenschaft `ScrollBars` einrichten können. Ist `WordWrap` auf `true` eingestellt (Voreinstellung), bricht die Textbox den Text am rechten Rand automatisch um, ohne allerdings eine neue Zeile im Text zu erzeugen. Wenn Sie eine horizontale Scrollbar verwenden, sollten Sie `WordWrap` auf `false` einstellen.

Über die Eigenschaft `AcceptsReturn` können Sie festlegen, ob eine mehrzeilige Textbox die Return-Taste als Zeilenumbruch auswertet oder nicht wenn das Formular einen Default-Button besitzt. Auf Formularen, die einen Default-Button besitzen (der eigentlich über die Return-Taste automatisch betätigt wird), ist es oft ein Problem, wenn mehrzeilige Textboxen die Return-Taste für sich beanspruchen. Steht der Cursor in der Textbox, führt ein Return nicht mehr dazu, dass der Default-Button betätigt wird. `AcceptsReturn` ist deswegen per Voreinstellung auch auf `false` eingestellt. Der Anwender kann einen Zeilenumbruch dann aber immer noch über Strg ↵ eingeben.

Ähnliches gilt für die Eigenschaft `AcceptsTab`. Ist diese bei einer mehrzeiligen Textbox auf `true` eingestellt, kann der Anwender einen Tabulator eingeben. Die Betätigung der Tabulatortaste führt dann nicht dazu, dass der Cursor auf das nächste Steuerelement wechselt.

Mehrzeilige Textboxen können Sie einfach über die `Text`-Eigenschaft auswerten. Die Zeilenumbrüche sind als "\r\n" im Text enthalten. Wenn Sie die einzelnen Zeilen auswerten wollen, können Sie aber auch die `Lines`-Eigenschaft verwenden, die ein String-Array zurückgibt:

```
foreach (string line in txtDemo.Lines)
{
   MessageBox.Show(line);
}
```

`Lines` wertet scheinbar nur die `Text`-Eigenschaft aus und erzeugt aus den gefundenen Zeilen ein Array. Ein Hinzufügen von Zeilen ist über `Lines` (leider) nicht möglich. Über diese Eigenschaft können Sie einer Textbox aber ein existierendes String-Array zuweisen:

```
string[] stringArray = {"Das ist", "ein",
   "String-Array"};
txtDemo.Lines = stringArray;
```

Die Textbox kopiert die Strings des übergebenen Arrays in die `Text`-Eigenschaft. Eine Änderung des Textes in der Textbox beeinflusst nicht das Array.

Wenn Sie Text dynamisch eintragen wollen, können Sie dazu auch die `Text`-Eigenschaft oder die `AppendText`-Methode verwenden. Kennzeichnen Sie Zeilenumbrüche dann durch `"\r\n"`:

```
txtDemo.Text = "Ich bin";
txtDemo.Text += "\r\neine"
txtDemo.AppendText("\r\nTextbox");
```


Selektion auswerten

Wenn Sie auf die aktuelle Selektion zugreifen wollen, erhalten Sie den selektierten Text in der Eigenschaft `SelectedText`. Über `SelectionStart` und `SelectionLength` können Sie die Selektion auch neu definieren.

Tipps und Tricks

Text immer automatisch selektieren

Die Textbox merkt sich die aktuelle Auswahl und stellt diese wieder her, wenn sie den Fokus erhält. Wenn Sie in diesem Fall aber immer den gesamten Text selektieren wollen, müssen Sie im `Enter`-Ereignis die `SelectAll`-Methode aufrufen:

```
private void txtDemo_Enter(object sender,
   System.EventArgs e)
```

```
{
   TextBox txt = (TextBox)sender;
   txt.SelectAll();
}
```

Nur Zahlen zulassen

Wenn Sie nur Zahlen als Eingabe zulassen wollen, programmieren Sie im `KeyPress`-Ereignis. Da Sie das Ereignis auch als behandelt kennzeichnen können, können Sie andere Eingaben einfach ignorieren. Sie müssen aber darauf achten, die Backspace-Taste auch zuzulassen:

```
private void txtDemo_KeyPress(object sender,
   System.Windows.Forms.KeyPressEventArgs e)
{
   /* Nur Zahlen und Backspace zulassen */
   if ((e.KeyChar < '0' || e.KeyChar > '9') &&
     (e.KeyChar != 8) )
   {
      e.Handled = true;
      MessageBox.Show("Bitte geben Sie nur " +
         "Zahlen ein");
   }
}
```

7.2.9 CheckBox und RadioButton

Die `CheckBox` (mein Präfix: *chk*) und der `RadioButton` (mein Präfix: *rb*) können vom Anwender betätigt werden, um den Wert zwischen ein- und ausgeschaltet zu ändern. Radiobuttons werden immer gruppenweise angelegt. Innerhalb einer Gruppe kann der Anwender nur einen Radiobutton einschalten, ein eventuell zuvor eingeschalteter wird dann automatisch ausgeschaltet. Bei Checkboxen kann der Anwender immer auch mehrere gleichzeitig einschalten.

Der Container definiert die Gruppe von Radiobuttons. Alle Radiobuttons auf einem Container gehören zur selben Gruppe. Eine einzelne Gruppe können Sie einfach auf einem Formular anlegen. Wenn Sie mehrere Gruppen erzeugen wollen, müssen Sie diese auf einem anderen Container, wie z. B. der `GroupBox` anlegen.

Wie bei allen Steuerelementen, die einen Text besitzen, definiert die Eigenschaft `Text` die Beschriftung. Sie können aber auch über `Image`, `ImageList` und `ImageIndex` ein Bild darstellen. Die boolesche Eigenschaft `Checked` bestimmt, ob das Steuerelement ein- oder ausgeschaltet ist.

Die Checkbox kennt neben dem Zustand *Ein* und *Aus* noch einen Zwischenzustand. Wenn die Eigenschaft `ThreeState true` ist, kann auch der Anwender diesen Zwischenzustand einschalten. Ist `ThreeState false` (Voreinstellung), kann der Zwischenzustand nur im Programm eingeschaltet werden. Setzen Sie dazu die Eigenschaft `CheckState` auf `CheckState.Indeterminate`. Wie Sie es wahrscheinlich von verschiedenen Optionen beim Installieren von Anwendungen kennen, repräsentiert der Zwischenzustand weder ein- noch ausgeschaltet.

Wie beim Label können Sie durch einfaches Voranstellen eines »&« vor einem Zeichen in der Beschriftung einen Shortkey einstellen.

Interessant ist, dass Sie über die Eigenschaft `Apperance` aus den beiden Steuerelementen einen Umschalter machen können.

7.2.10 ListBox

Das `ListBox`-Steuerelement (mein Präfix: *lbx*) zeigt eine Reihe von Einträgen in einer Liste an, aus denen der Anwender einen oder auch mehrere auswählen kann. Die Liste kann ein- und mehrspaltig dargestellt werden und erlaubt die Auswahl einzelner oder auch mehrerer Einträge.

Füllen der Liste

Die Listbox verwaltet ihre Liste über die Eigenschaft `Items`, die eine Auflistung vom Typ `ListBox.ObjectCollection` ist. Diese Auflistung können Sie im Eigenschaftenfenster füllen. Das dynamische Füllen im Programm ist bei einer Listbox aber wohl eher die Regel.

Über die `Add`-Methode dieser Auflistung können Sie der Liste beliebige Objekte hinzufügen. Die Elemente der Liste besitzen den Typ `object`. Die Liste kann also alle möglichen Typen verwalten. Die Listbox gibt den String, den die `ToString`-Methode der gespeicherten Objekte zurückgibt, in der Liste aus.

Häufig werden Listboxen für die Speicherung von Strings verwendet:

```
lbxHobbies.Items.Add("Snowboarden");
lbxHobbies.Items.Add("Inlinern");
lbxHobbies.Items.Add("Motorradfahren");
```

Die Anzeige von im Programm gespeicherten Objekten ist in der Praxis aber auch sehr wichtig. Objekte können Sie ebenfalls einfach über die `Add`-Methode hinzufügen. Objekte werden im Programm aber auch häufig über Arrays oder Auflistungen verwaltet. Über die `AddRange`-Methode können Sie der Listbox ein `object`-Array hinzufügen, das Sie beispielsweise der `ToArray`-Methode einer Auflistung entnehmen können.

Das Beispiel zum Füllen der Listbox mit Objekten erzeugt eine Auflistung, fügt einige Personen-Objekte an und übergibt diese Auflistung dann einer Listbox. Die Person-Klasse ist für das Beispiel so implementiert:

```
public class Person
{
   public int ID;
   public string FirstName;
   public string SurName;

   public Person(int id, string firstName,
      string surName)
   {
      this.ID = id;
      this.FirstName = firstName;
      this.SurName = surName;
   }

   /* Die ToString-Methode wird überschrieben,
      * damit die Listbox die Objekte korrekt
      * darstellen kann */
   public override string ToString()
   {
      return this.FirstName + " " + this.SurName;
   }
}
```

Der folgende Quellcode erzeugt eine Auflistung und füllt diese mit einigen Personen-Objekten:

```
System.Collections.ArrayList alPersons =
   new System.Collections.ArrayList();
alPersons.Add(new Person(1001"Philo","Dufresne"));
alPersons.Add(new Person(1002,"Joan","Fine"));
alPersons.Add(new Person(1003,"Lexa","Thatcher"));
alPersons.Add(new Person(1004,"Harry","Gant"));
alPersons.Add(new Person(1005,"Morris","Katzenstein"));
```

Dann wird die Auflistung einfach an die Liste angehangen:

```
lbxDemo.Items.AddRange(alPersons.ToArray());
```

Die Listbox zeigt nun für jedes Objekt das Resultat der `ToString`-Methode an (Abbildung 7.7).

Bild 7.7: Eine Listbox, die den Inhalt einer Auflistung anzeigt

Die `Add`-Methode hängt die Einträge immer hinten an die Liste an, sofern diese nicht über die `Sorted`-Eigenschaft sortiert wird. Über die `Insert`-Methode können Sie Objekte auch an eine bestimmte Position in der Liste anfügen:

```
Person p = new Person(1006, "Vanna", "Domingo")
/* an die 2. Position einfügen */
lbxDemo.Items.Insert(1, p);
```

Optimieren der Füllung mit vielen Einträgen

Wenn Sie die Listbox mit vielen Einträgen füllen, sollten Sie vor dem Füllen die `BeginUpdate`-Methode und danach die `EndUpdate`-Methode aufrufen. Die Listbox wird dann nicht immer wieder neu gezeichnet und deshalb schneller gefüllt.

Sortierte Listen

Wenn Sie die `Sorted`-Eigenschaft der Listbox auf `true` setzen, wird die Liste immer automatisch bezogen auf ihre Einträge sortiert.

Löschen von Einträgen

Vor dem (Wieder-)Füllen der Liste ist es häufig notwendig, die alten gespeicherten Einträge zu löschen. Dazu verwenden Sie die `Clear`-Methode der `Items`-Eigenschaft:

```
lbxDemo.Items.Clear();
```

Mit der `RemoveAt`-Methode können Sie gezielt einzelne Einträge löschen.

Mehrspaltige Listboxen

In der Eigenschaft `MultiColumn` können Sie mit `true` festlegen, dass die Listbox die Einträge nicht nur untereinander, sondern auch nebeneinander darstellt. Die Listbox legt dann nur so viele Zeilen an, wie in den sichtbaren Bereich der Listbox passen und bricht die Zeilen in die nächste Spalte um. Die Breite der Spalten können Sie über die Eigenschaft `ColumnWidth`-Eigenschaft bestimmen. Abbildung 7.8 zeigt eine mehrspaltige Listbox mit den Daten der Auflistung aus dem letzten Beispiel.

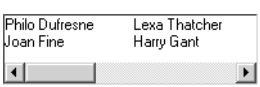

Bild 7.8: Eine mehrspaltige Listbox.

Auswahlmöglichkeit festlegen

In der Eigenschaft `SelectionMode` können Sie festlegen, dass der Benutzer keine Einträge auswählen kann (`None`), oder nur einzelne (`One`), oder auch mehrere (`MultiSimple` und `MultiExtended`). Der Unterschied zwischen `MultiSimple` und `MultiExtended` ist, dass der Anwender bei einer einfachen Mehrfachauswahl die einzelnen Einträge einfach über die Maus oder die Leertaste aktivieren und deaktivieren kann. Bei der erweiterten Mehrfachauswahl muss der Anwender die Maus in Verbindung mit ⇧ und Strg verwenden, um mehrere Einträge auszuwählen.

Einträge vorselektieren

Wenn Sie eine Listbox neu füllen, ist zunächst kein Eintrag selektiert. Um einzelne Einträge vorzuselektieren, setzen Sie bei einer Einfachauswahl-Listbox `SelectedIndex` auf den Index des Eintrags:

```
/* Ersten Eintrag vorselektieren */
lbxDemo.SelectedIndex = 0;
```

Bei einer Mehrfachauswahl-Listbox können Sie `SelectedIndex` verwenden, wenn Sie nur einen Eintrag selektieren wollen. Wollen Sie aber mehrere Einträge selektieren, müssen Sie dazu die `SetSelected`-Methode der Listbox verwenden, der Sie im ersten Argument den Index des Elements übergeben und im zweiten Argument `true`, um das Element zu selektieren:

```
lbxDemo.SetSelected(2,true);
lbxDemo.SetSelected(3,true);
```

Wenn Sie im zweiten Argument `false` übergeben, setzen Sie eine eventuelle Selektierung zurück.

Die allerwichtigsten Ereignisse

Über das `Click`-Ereignis können Sie darauf reagieren, dass der Anwender auf einen Eintrag klickt, über `DoubleClick` werten Sie einen Doppelklick aus. Eine Änderung der aktuellen Selektion (die auch über die Tastatur erreicht werden kann) werten Sie über das Ereignis `SelectedIndexChanged` aus. Einige spezielle Ereignisse wie `DrawItem` und `MeasureItem` erlauben Ihnen, die Einträge in der Listbox selbst zu zeichnen.

Auswertung der ausgewählten Einträge

In einer Einfachauswahl-Listbox können Sie einfach die Eigenschaft `Text` auswerten, um den Text des aktivierten Eintrags zu erhalten. `Text` ist leer, wenn kein Eintrag ausgewählt ist.

Sie können aber auch den Index des ausgewählten Eintrags über die Eigenschaft `SelectedIndex` auswerten. Wenn diese Eigenschaft `-1` zurückgibt, ist kein Eintrag ausgewählt. Wenn ein Eintrag ausgewählt ist, können Sie über `SelectedItem` auf das gespeicherte Objekt zugreifen. Das folgende Beispiel geht davon aus, dass Person-Objekte in der Listbox gespeichert sind:

```
if (lbxDemo.SelectedIndex > -1)
{
   Person p = (Person)lbxDemo.SelectedItem;
   MessageBox.Show("Person mit ID " + p.ID +
      " wurde ausgewählt: " + p.FirstName +
      " " + p.SurName);
}
else
{
   MessageBox.Show("Kein Objekt selektiert");
}
```

Bei einer Mehrfachauswahl-Listbox müssen Sie etwas anders vorgehen. Hier können ja mehrere Einträge selektiert sein. Für den direkten Zugriff auf die Objekte werten Sie dann die Eigenschaft `SelectedItems` aus, die eine Auflistung vom Typ `object` ist:

```
if (lbxDemo.SelectedItems.Count > 0)
{
   foreach (object o in lbxDemo.SelectedItems)
   {
      Person p = (Person)o;
      MessageBox.Show(p.ID + ": " + p.FirstName +
         " " + p.SurName);
   }
}
else
{
   MessageBox.Show("Kein Objekt selektiert");
}
```

Wenn Sie lediglich die *Indizes* der selektierten Objekte benötigen, können Sie die `SelectedIndices`-Eigenschaft auswerten, die ebenfalls eine Auflistung ist.

Suchen in einer Listbox

Über die Methoden `FindString` und `FindStringExact` können Sie in einer Listbox sehr schnell suchen. Beiden Methoden übergeben Sie im ersten Argument den Suchstring und im zweiten den Index, ab dem gesucht werden soll. Die Rückgabe ist der Index des gefundenen Ein-

trags oder -1, wenn nichts gefunden wurde. `FindString` sucht einen Eintrag, der mit dem Suchstring beginnt, `FindStringExact` sucht einen Eintrag, der dem Suchstring genau entspricht. Eine häufige Anwendung ist die Suche nach Listbox-Einträgen über eine Textbox.

Bild 7.9: Suchen in einer Liste über eine Textbox

Wenn der Anwender den Text ändert, sucht das Programm nach dem ersten passenden Eintrag in der Listbox und selektiert diesen. Die Programmierung erfolgt in der `KeyPress`-Ereignisbehandlungsmethode der Textbox:

```
private void txtFind_TextChanged(object sender,
   System.EventArgs e)
{
   lbxDemo.SelectedIndex =
      lbxDemo.FindString(txtFind.Text,0);
}
```

7.2.11 CheckedListBox

Das `CheckedListBox`-Steuerelement (mein Präfix: *clb*) ist eine Listbox, die ihre Einträge als Checkbox anzeigt, sodass der Anwender beim Selektieren die Checkbox auswählen kann. Da das `CheckedListBox`-Steuerelement direkt vom `ListBox`-Steuerelement abgeleitet ist, ist die Programmierung zur Listbox nahezu identisch. Die CheckedListbox lässt aber grundsätzlich eine Mehrfachauswahl zu, wenn `SelectionMode` nicht auf `None` steht. Zur Auswertung der ausgewählten Einträge verwenden Sie die Eigenschaften `CheckedIndices` und `CheckedItems`, die mit ihren Vettern bei der Listbox identisch sind. Über die Eigenschaft `CheckOnClick` können Sie festlegen, ob ein Eintrag direkt beim Anklicken selektiert wird, oder ob das Selektieren nach dem Anklicken separat erfolgen muss (was die Voreinstellung ist).

7.2.12 ComboBox

Das `ComboBox`-Steuerelement (mein Präfix: *cbx*) ist der Listbox sehr ähnlich. Der erste erkennbare Unterschied ist, dass die Liste zunächst nicht sichtbar ist, sondern (normalerweise) erst aufklappt, wenn der Anwender auf den Pfeil rechts klickt.

Die Programmierung erfolgt eigentlich genau wie die der Listbox, in vielen Fällen können Sie eine Listbox einfach gegen eine Combobox austauschen. Ein wesentlicher Unterschied ist, dass die Combobox mit dem Wert `Simple` oder `DropDown` in der Eigenschaft `DropDownStyle` ein Textfeld besitzt, in das der Anwender unabhängig von der Liste Text eingeben kann. Der andere wesentliche Unterschied ist, dass die Combobox keine Mehrfachauswahl ermöglicht.

Über die Eigenschaft `DropDownStyle` bestimmen Sie, ob der Anwender im Textfeld der Combobox selbst Text eingeben kann und ob die Combobox einen Pfeil anzeigt, über den die Liste aufgeklappt werden kann. Die folgenden Werte sind möglich:

- `Simple`: Die Combobox sieht aus wie eine einfache Textbox. Der Anwender kann Text eingeben. Er kann aber auch über die Cursortasten einen Eintrag aus der Liste auswählen. Die Auswahl wird in das Textfeld geschrieben.
- `DropDown`: Die Combobox sieht aus wie eine Textbox mit einem Pfeil an der rechten Seite. Der Anwender kann Text eingeben und einen Eintrag der Liste über die Cursortasten auswählen. Alternativ kann er über den Pfeil die Liste aufklappen und dort einen Eintrag auswählen.
- `DropDownList`: Wie `DropDown`, die Combobox erlaubt aber keine Texteingabe.

Gefüllt wird eine Combobox genau wie eine Einfachauswahl-Listbox. Bei der Auswertung müssen Sie beachten, dass die Eigenschaft `Text` beim Dropdown-Stil `Simple` oder `DropDown` die Texteingabe des Anwenders speichert. Entspricht die Texteingabe keinem Listeneintrag, gibt die Eigenschaft `SelectedIndex` -1 zurück. Sie können also bei diesen Dropdown-Stilen den Text separat auswerten. Die Auswertung könnte also beispielsweise so aussehen:

```
if (cbxDemo.SelectedIndex > -1)
{
   MessageBox.Show("Eintrag an Index " +
      cbxDemo.SelectedIndex + " ausgewählt: " +
      cbxDemo.Text);
}
else if (cbxDemo.Text != "")
{
   MessageBox.Show("Kein Eintrag ausgewählt. " +
      "Text eingegeben: " + cbxDemo.Text);
}
else
{
   MessageBox.Show("Kein Eintrag ausgewählt. " +
      "Kein Text eingegeben.");
}
```

Beim Stil `DropDownList` entspricht der Text immer dem aktuell selektierten Eintrag.

7.2.13 Container: GroupBox und Panel

`GroupBox` (mein Präfix: *gbx*) und `Panel` (mein Präfix: *pnl*) sind eher passive Steuerelemente, die als Container für andere Steuerelemente dienen. Wenn andere Steuerelemente auf einer Groupbox oder einem Panel platziert werden, hat das mehrere Vorteile:

- Optionsschalter über einen Container werden zu einer Gruppe zusammengefasst;
- Alle auf einem Container platzierten Steuerelemente können im Entwurf über die Selektion des Containers in die Zwischenablage kopiert werden;
- Durch Verschieben des Containers werden automatisch auch die enthaltenen Steuerelemente mit verschoben.

Die Groupbox kann (über die Eigenschaft `Text`) eine Beschriftung im oberen Bereich anzeigen, und zeigt immer einen dreidimensionalen Rahmen an, was beim Panel nicht der Fall ist. Dafür besitzt das Panel die Eigenschaft `AutoScroll`, über die Sie einstellen können, dass das Panel automatisch seinen Inhalt über Scroll-Leisten scrollbar macht,

wenn dieser nicht komplett in den sichtbaren Bereich passt. Das Panel kann zudem über die Eigenschaft `BorderStyle` mit einem Rahmen ausgestattet werden.

7.2.14 PictureBox

Das `PictureBox`-Steuerelement (mein Präfix: *pbx*) können Sie zur Ausgabe von Bildern oder auch zum Zeichnen (mit Hilfe der Grafikklassen) verwenden.

Ein Bild wird in der Eigenschaft `Image` verwaltet. Die Eigenschaft `SizeMode` bestimmt, wie das Bild bzw. die Picturebox in der Größe angepasst wird:

- `Normal`: Weder das Bild noch die Picturebox werden in der Größe verändert. Das Bild wird ausgehend von der linken oberen Ecke der Picturebox positioniert.
- `StretchImage`: Das Bild wird auf die Größe der Picturebox skaliert (und dabei gestreckt oder gestaucht).
- `AutoSize`: Die Picturebox wird auf die Größe des Bildes skaliert.
- `CenterImage`: Weder das Bild noch die Picturebox werden in der Größe verändert. Das Bild wird in der Mitte der Picturebox positioniert.

Über die `BorderStyle`-Eigenschaft können Sie einen Rahmen definieren. In der Laufzeit können Sie ein Bild aus einer Datei oder einer Ressource laden. Das Laden aus einer Datei wurde bereits auf der Seite 323 behandelt.

7.2.15 Timer

Der Timer (mein Präfix: *tmr*) ist eigentlich gar kein Steuerelement, sondern eine Komponente, taucht aber trotzdem in der Toolbar als Steuerelement auf. Der Timer generiert in regelmäßigen Abständen das Ereignis `Tick`, in dessen Ereignisbehandlungsmethode Sie zeitgesteuerte Abläufe einrichten können. In der Eigenschaft `Intervall` legen Sie das Intervall für die Ticks des Timer im Millisekunden fest. Eine Einstellung von 1000 bewirkt beispielsweise, dass das `Tick`-Ereignis alle 1000 ms generiert wird. Wenn Sie die Eigenschaft `Enabled` auf `true` setzen, beginnt der Timer zu laufen, mit `Enabled=false` können Sie den Timer wieder anhalten.

7.2.16 Die Datei-Dialoge am Beispiel des OpenFileDialog

Die Komponenten `OpenFileDialog` (mein Präfix: *ofd*), `SaveFileDialog` (mein Präfix: *sfd*), `FontDialog` (mein Präfix: *fd*), `ColorDialog` (mein Präfix: *cd*) und `PrintDialog` (mein Präfix: *pd*) sind enorm hilfreich, wenn Sie den Anwender eine Datei, eine Schriftart, eine Farbe oder Druckeinstellungen auswählen lassen wollen. Über diese Komponenten können Sie recht einfach die Standarddialoge des Betriebssystems verwenden. Die Komponenten sind einfach anzuwenden. Ich zeige dies am Beispiel der `OpenFileDialog`-Komponente, die prinzipiell identisch ist mit der Komponente `SaveFileDialog`.

In den Buchbeispielen zu diesem Kapitel finden Sie ein Beispiel, das alle Dialoge demonstriert.

OpenFileDialog

Vor dem Öffnen eines Datei-Dialogs sollten Sie in der Eigenschaft `Filter` einen oder mehrere Dateifilter einstellen. Diese Dateifilter erscheinen nachher in der Dateityp-Liste des Dialogs und schränken die Ansicht der Dateien entsprechend ein. Den Filter definieren Sie mit einem String, der folgendermaßen aussieht:

```
"FilterName1|Maske1[|Filtername2|Maske2][...]"
```

Ein typischer Filter sieht z. B. so aus:

```
"Bilddateien|*.bmp;*.gif;*.jpg;*.png|Alle Dateien|*.*"
```

In der Eigenschaft `FilterIndex` können Sie den Filter angeben, der beim Öffnen aktiviert ist. Geben Sie beim Beispielfilter oben z. B. `1` an, ist der Filter »Alle Dateien« aktiv.

In die Eigenschaft `DefaultExt` sollten Sie eine Dateiendung schreiben, die dann vom Dialog verwendet wird, wenn der Anwender nur den Basisnamen der Datei ohne Endung angibt. Geben Sie die Endung immer mit dem Punkt an. Die Eigenschaft `FileName` können Sie mit einem Dateinamen beschreiben, der beim Öffnen des Dialogs voreingestellt sein soll. In `InitialDirectory` können Sie das Verzeichnis einstellen, in dem der Dialog startet.

Um festzulegen, dass der Anwender nur Dateien auswählen kann, die existieren, können Sie `CheckFileExists` auf `true` setzen. Der Dialog lässt dann das Schließen über den OK-Schalter nur zu, wenn die angegebene Datei existiert. Wenn Sie nur überprüfen lassen wollen, dass der angegebene Pfad existiert, setzen Sie `CheckPathExists` auf `true`.

Schließlich öffnen Sie den Dialog über dessen `OpenDialog`-Methode. Diese Methode gibt den Wert `DialogResult.OK` zurück, wenn der Anwender den OK-Schalter betätigt hat.

Das folgende Beispiel initialisiert einen `OpenFileDilaog`, öffnet dann den Dialog und gibt den ausgewählten Dateinamen in einer Messagebox aus:

```
/* Filter setzen */
ofdDemo.Filter = "Bilddateien|*.bmp;*.gif;*.jpg;" +
   "*.png|" Alle Dateien|*.*";
/* Default-Endung definieren (falls der Anwender
   * einen Dateinamen ohne Endugn angibt) */
ofdDemo.DefaultExt = ".bmp";
/* Dateinamen voreinstellen */
ofdDemo.FileName = "Galaxy.bmp";
/* Startverzeichnis definieren */
ofdDemo.InitialDirectory = @"C:\Winnt";
/* Festlegen, dass der Dialog dafür sorgen
 * soll, dass nur Dateien ausgewählt werden
 * können, die auch existieren */
ofdDemo.CheckFileExists = true;

/* Dialog anzeigen */
if (ofdDemo.ShowDialog() == DialogResult.OK)
{
   string filename = ofdDemo.FileName;
   MessageBox.Show(filename);
}
else
{
   MessageBox.Show("Abbruch des Anwenders";
}
```

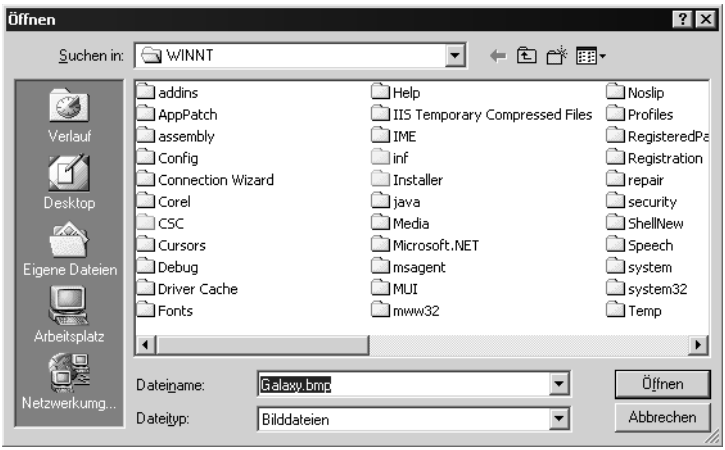

Bild 7.10: Der durch den OpenFileDialog geöffnete Datei-Öffnen-Dialog unter Windows 2000

Wenn Sie die Eigenschaft `MultiSelect` auf `true` setzen, kann der Anwender mehrere Dateien auswählen. Bei der Auswertung müssen Sie dann die Eigenschaft `FileNames` verwenden, die ein String-Array zurückgibt.

7.3 Überprüfen von Eingaben

Zum Überprüfen von Eingaben (Validieren) bieten Ihnen die Steuerelemente Hilfestellung über Eigenschaften und Ereignisse. Jedes Steuerelement, das den Eingabefokus erhalten kann, besitzt eine Eigenschaft `CausesValidation` und ein Ereignis `Validating`. Im `Validating`-Ereignis können Sie Validierungen vornehmen. Dieses Ereignis wird immer dann aufgerufen, wenn der Benutzer ein Steuerelement verlassen und zu einem Steuerelement wechseln will, dessen `CausesValidation`-Eigenschaft auf `true` steht. Wenn Sie erkennen, dass der Benutzer einen inkorrekten Wert eingegeben hat, setzen Sie einfach das `Cancel`-Argument des zweiten Arguments des `Validating`-Ereignisses auf `true`. Dann bleibt der Eingabecursor im aktuellen Steuerelement.

Mit dieser Technik können Sie eine Validierung der einzelnen Eingaben direkt beim Verlassen des Steuerelements erreichen. Dass Sie `CausesValidation` auch auf `false` einstellen können, macht Sinn: Um dem Benutzer einen Abbruch zu ermöglichen, stellen Sie diese Eigenschaft beim Abbrechen-Schalter auf `false`. Ein Abbruch bewirkt dann, dass kein `Validating`-Ereignis aufgerufen wird, weil ja auch nicht validiert werden muss.

Das folgende Beispiel basiert auf einem Formular, in dem der Anwender seinen Namen eingeben soll:

```
private void txtName_Validating(object sender,
   System.ComponentModel.CancelEventArgs e)
{
   /* Validieren */
   if (txtName.Text == "")
   {
      MessageBox.Show("Geben Sie einen Namen ein");
      /* Das Verlassen des Steuerelements abbrechen */
      e.Cancel = true;
   }
}

private void btnCancel_Click(object sender,
   System.EventArgs e)
{
   this.Close();
}

private void btnOK_Click(object sender,
   System.EventArgs e)
{
   /* Irgendetwas mit dem Namen machen */
   /* ... */
   this.Close();
}
```

Sinnvoll ist das Ganze nur dann, wenn Sie direkt beim Verlassen eines Steuerelements validieren wollen. Wenn Sie die Eingaben nur

dann überprüfen wollen, wenn der Anwender den OK-Schalter betätigt, verwenden Sie besser das `Click`-Ereignis dieses Schalters.

Wenn Sie die korrekte Eingabe auch dann ermöglichen wollen, wenn der Anwender das Formular nicht über einen Schalter schließt, können Sie mit der `Validate`-Methode des Formulars erreichen, dass auch das Steuerelement, das zuletzt den Fokus hatte, überprüft wird. Dies wird allerdings nur selten notwendig sein, da Sie zur Bestätigung der Eingaben normalerweise einen OK-Schalter auf dem Formular ablegen und bei diesem `CausesValidation` auf `true` steht.

7.4 Formulare

Die meisten Projekte kommen nicht mit nur einem Formular aus. Oft müssen in einem Programm andere Formulare für die verschiedensten Zwecke geöffnet werden. Sie können in einem Projekt deshalb mehrere Formulare verwalten. Ich zeige hier, wie Sie diese einstellen, öffnen und wie Sie Eingaben auswerten, die in anderen Formularen vorgenommen wurden.

7.4.1 Die wichtigsten Klassenelemente der Form-Klasse

Die wichtigsten Eigenschaften

Alle Formulare sind zumindest von der Klasse `Form` abgeleitet und besitzen deshalb die Elemente dieser Klasse. Die wichtigsten stelle ich hier vor. Die Klasse Form ist übrigens indirekt von `Control` abgeleitet. Ein Formular besitzt deswegen auch die Elemente dieser Klasse (die ich hier nicht mehr weiter aufführe). Tabelle 7.5 listet die wichtigsten Eigenschaften auf.

Eigenschaft	Beschreibung
`AcceptButton`	In dieser Eigenschaft können Sie einen Button angeben, der automatisch betätigt wird, wenn der Anwender die ↵-Taste betätigt.
`AutoScroll`	Diese Eigenschaft definiert, dass das Formular automatisch Scroll-Leisten anzeigt und damit seinen Inhalt scrollbar macht, wenn sein Inhalt größer ist als der sichtbare Bereich.

Eigenschaft	Beschreibung
CancelButton	In dieser Eigenschaft können Sie einen Button angeben, der automatisch betätigt wird, wenn der Anwender die Esc-Taste betätigt.
ControlBox	Legt fest, ob das Fenster ein Systemmenüfeld und den Schließen-Schalter in der Titelleiste besitzt. Eventuell vorhandene Minimieren- und Maximieren-Schalter werden auch ausgeblendet, wenn ControlBox false ist.
DockPadding	bestimmt den Rahmen zwischen Formular und Steuerelementen, die über deren Dock-Eigenschaft an das Formular angedockt werden
DrawGrid	definiert, ob das Formular im Entwurf das Gitternetz anzeigt
FormBorderStyle	legt den Rahmen und die Titelleiste fest. Die möglichen Werte sind: None (kein Rahmen, keine Titelleiste, keine Größenänderung), FixedSingle (einfacher Rahmen, keine Größenänderung über die Maus, Minimieren und Maximieren möglich), Fixed3D (wie FixedSingle, 3D-Rahmen), FixedDialog (wie FixedSingle, normaler Rahmen), Sizeable (normaler Rahmen, Größenveränderung möglich), FixedToolWindow (normaler Rahmen, keine Größenveränderung möglich, Titelleiste verkleinert) und SizeableToolWindow (wie FixedToolWindow, Größenveränderung möglich).
GridSize	legt die Abstände der Linien für das Gitternetz fest
HelpButton	Wenn das Formular keinen Maximieren- und Minimieren-Schalter besitzt, können Sie über HelpButton einen kleinen Hilfeschalter neben dem Schließen-Schalter in der Titelleiste anzeigen lassen. Die Betätigung dieses Schalters können Sie über das Ereignis HelpRequested auswerten.
Icon	definiert das Icon des Formulars
IsMDIContainer	definiert, ob das Formular ein MDI-Container ist. MDI-Formulare werden auf der Seite kurz besprochen.

Eigenschaft	Beschreibung
KeyPreview	Wenn Sie hier true eingeben, werden bei einer Tastenbetätigung immer erst die Tastaturereignisse des Formulars aufgerufen, bevor diese für das aktive Steuerelement aufgerufen werden. Damit können Sie allgemeine Tastaturbelegungen für das gesamte Formular programmieren.
MaximizeBox, MinimizeBox	Diese Eigenschaften definieren, ob der Maximieren- und den Minimieren-Schalter in der Titelleiste sichtbar sind.
MaximumSize, MinimumSize	Diese Eigenschaften definieren die maximale bzw. die minimale Größe auf die der Anwender das Formular vergrößern bzw. verkleinern kann.
MDIParent	Diese nur in der Laufzeit verfügbare Eigenschaft macht ein Formular zu einem MDI-Child, indem das MDI-Parent-Formular angegeben wird.
Menu	In dieser Eigenschaft geben Sie das Menü des Formulars an. Menüs werden ab Seite 368 besprochen.
Opacity	Über Opacity können Sie ein Formular durchsichtig schalten. Ein Wert von 50 bedeutet z. B., dass das Formular zu 50% durchsichtig ist.
ShowInTaskbar	definiert, ob das Formular in der Taskleiste angezeigt wird.
SnapToGrid	definiert für den Entwurf, ob Steuerelemente am Gitternetz ausgerichtet werden.
StartPosition	legt fest, wo das Formular auf dem Bildschirm erscheint. Die folgenden Einstellungen sind möglich: Manual (das Formular erscheint an der durch Location definierten Position), CenterScreen (das Formular erscheint in der Mitte des Bildschirms), WindowsDefaultLocation (das Formular erscheint an der Stelle, die Windows standardmäßig vorsieht), WindowsDefaultBounds (das Formular erscheint an der Stelle, die Windows standardmäßig vorsieht und besitzt die Größe, die Windows standardmäßig bestimmt) und CenterParent (das Formular erscheint in der Mitte seines Parent).

Eigenschaft	Beschreibung
`Text`	definiert die Beschriftung
`TopMost`	Über diese Eigenschaft können Sie dafür sorgen, dass das Formular über allen anderen Windows-Fenstern angezeigt wird, auch wenn es gerade nicht aktiv ist.
`TransparencyKey`	Hier können Sie eine Farbe angeben, die das Formular in transparente Bereiche umwandelt. Der Anwender kann durch die transparenten Bereiche hindurchschauen. Mausbetätigungen werden an die darunter liegenden Fenster weitergegeben. Wenn die Hintergrundfarbe des Formulars dieselbe Farbe besitzt, ist der gesamte Innenbereich des Formulars durchsichtig. Mit gezieltem Zeichnen auf dem Formular können Sie einzelne Bereiche transparent schalten. Wenn Sie dem Formular über `BackGroundImage` ein Bild zuweisen, werden die Teile dieses Bildes transparent dargestellt, die die »transparente Farbe« besitzen.
`WindowState`	Legt fest, ob das Formular beim Öffnen normal (d. h. in der durch `Size` definierten Größe), minimiert oder auf dem Bildschirm maximiert erscheint

Tabelle 7.5: Die wichtigsten Eigenschaften eines Formulars

Die wichtigsten Methoden der Form-Klasse

Methode	Beschreibung
`void AddOwnedForm(` `Form` *`ownedForm`*`)`	Wenn das Formular als »Besitzer« anderer Formulare dienen soll, können Sie diese über `AddOwnedForm` der Auflistung der besitzten Formulare hinzufügen. Formulare, die anderen Formularen »gehören«, werden automatisch minimiert und geschlossen, wenn der Besitzer-Formular minimiert oder geschlossen wird und werden immer hinter dem Besitzer-Formular angezeigt.

Methode	Beschreibung
`void Close()`	schließt ein Formular. Dabei werden alle Ressourcen freigegeben, die das Formular verwendet und das Formular zur Zerstörung durch den Garbage Collector freigegeben.
`void Hide()`	schließt ein Formular. Das Formular wird allerdings nur versteckt und bleibt im Arbeitsspeicher geladen. Alle Eingaben bleiben erhalten. Sie können auf diese im Programm auch dann zugreifen, wenn das Formular geschlossen ist. Das Formular kann über `Show` wieder angezeigt werden.
`void LayoutMdi(` `MdiLayout` *`value`*`)`	arrangiert die MDI-Child-Fenster einer MDI-Anwendung auf dem MDI-Formular (als Icon, kaskadierend, geteilt)

`DialogResult`
`ShowDialog(`
`[IWin32Window` *`owner`*`])`
	zeigt ein Formular als Dialog an. Ein Dialog muss vom Anwender bestätigt werden, damit das Programm weiterläuft.

`void Show()`	zeigt ein Formular normal an (nicht als Dialog). Der Anwender kann zwischen diesem und anderen Formularen der Anwendung wechseln. Das Programm läuft nach dem Öffnen weiter.

Tabelle 7.6: Die wichtigsten Methoden der Form-Klasse

Die wichtigsten Ereignisse der Form-Klasse

Ereignis	Beschreibung
`Activated`	wird aufgerufen, wenn das Formular vom Programm oder vom Anwender aktiviert wird (d. h., den Eingabefokus erhält). Dieses Ereignis wird häufig verwendet, um den Fokus auf ein bestimmtes Steuerelement zu setzen.
`Closed`	wird aufgerufen, nachdem ein Formular geschlossen wurde

Ereignis	Beschreibung
`Closing`	wird aufgerufen, wenn ein Formular dabei ist, sich zu schließen. Über die Eigenschaft `Cancel` des zweiten Arguments können Sie das Schließen abbrechen.
`Deactivate`	wird aufgerufen, wenn das Formular vom Anwender oder vom Programm deaktiviert wird (wenn der Anwender innerhalb der Anwendung ein anderes Formular aktiviert).
`Load`	wird aufgerufen, wenn ein Formular geladen wird. Wenn Sie ein Formular über `Hide` verstecken und dann wieder über `Show` anzeigen, wird `Load` nicht mehr aufgerufen (das Formular ist dann ja noch geladen). Hier können Sie Initialisierungen vornehmen. Der Konstruktor eines Formulars eignet sich dafür aber genauso. Das `Load`-Ereignis existiert wahrscheinlich nur deswegen, weil Visual Basic 6-Programmierer schon immer mit diesem Ereignis gearbeitet haben.
`MdiChild-Activate`	wird aufgerufen, wenn ein MDI-Child-Formular aktiviert wird

Tabelle 7.7: Die wichtigsten Ereignisse der Form-Klasse

7.4.2 Formulargrundlagen

Windows-Anwendungen mit und ohne Startformular

Windows-Anwendungen starten wie jede andere .NET-Anwendung auch mit der statischen Methode `Main`. Normalerweise wird innerhalb dieser Methode ein Formular erzeugt und geöffnet, das dann als Startformular der Anwendung gilt. Ausgehend von diesem Formular öffnen Sie dann über Buttons, Menüs oder ähnliche Steuerelemente andere Formulare.

Es gibt aber auch Windows-Anwendungen, die ähnlich einer Konsolenanwendung kein Startformular besitzen. Das Programm wird dann – genau wie bei einer Konsolenanwendung – innerhalb der Methode `Main` ausgeführt. Über das Argument der `Main`-Methode können Sie Befehlszeilenargumente auswerten, die der Anwender beim Starten

der Anwendung übergeben hat. Im Unterschied zu Konsolenanwendungen können Sie in einer Windows-Anwendung nicht auf die Konsole zurückgreifen, dafür aber innerhalb von `Main` auch (bei Bedarf) Formulare und Windows-Dialoge wie die Messagebox anzeigen.

Wann werden Windows-Anwendungen beendet?

Windows-Anwendungen ohne Startformular werden beendet, wenn die `Main`-Methode zu Ende ausgeführt ist. Windows-Anwendungen mit Startformular werden beendet, wenn das Startformular geschlossen wird. Alternativ könnten Sie eine Anwendung auch mit `Application.Exit` beenden. Das entspricht aber einem harten Abbruch und sollte nur in Ausnahmefällen verwendet werden.

Beim Aufruf der `Exit`*-Methode des* `Application`*-Objekts werden die* `Closing`*-Ereignisse der geöffneten Formulare nicht ausgeführt, in denen u. U. wichtige Aufräumarbeiten implementiert sind.*

MDI- und SDI-Anwendungen

Wenn Sie eine Anwendung entwickeln, müssen Sie sich entscheiden, ob Sie eine SDI(Single Document Interface)- oder MDI(Multiple Document Interfact)-Anwendung erzeugen wollen. Eine SDI-Anwendung startet mit einem normalen Formular. Weitere Formulare sind innerhalb solcher Anwendungen vom Startformular relativ unabhängig. Das Startformular kann zwar der Besitzer eines anderen Formulars sein, die anderen Formulare können aber frei auf dem Bildschirm platziert werden.

MDI-Anwendungen starten mit einem Formular, dessen Eigenschaft `IsMdiContainer` auf `true` eingestellt ist. Diese Formulare werden dann als MDI-Parent bezeichnet. Wenn Sie ausgehend von einem solchen Formular andere Formulare öffnen, können Sie das Startformular nun über die Eigenschaft `MDIParent` als MDI-Parent des anderen Formulars angeben. Das andere Formular wird dann zum MDI-Child. MDI-Child-Formulare werden immer innerhalb des MDI-Parent angezeigt. Zusätzliche Features, wie das einfache Einrichten eines *Fenster*-Menüpunkts, der automatisch alle geöffneten MDI-Childs auflistet, das halbautomatische Ausrichten der MDI-Childs über die Methode `LayoutMDI`, und die Tatsache, dass das Menü eines MDI-Childs in das

Menü des MDI-Parent eingeblendet wird, wenn das MDI-Child-Formular aktiv ist, erleichtern die Programmierung solcher Anwendungen. Microsoft Word ist übrigens ein gutes Beispiel für eine MDI-Anwendung: Das Hauptfenster von Word ist der MDI-Parent, die geöffneten Dokumente sind MDI-Childs.

Ich beschreibe im Buch nur den Umgang mit SDI-Formularen. Der Online-Artikel »MDI-Formulare«, den Sie auf der Website zum Buch finden, beschreibt den Umgang mit MDI-Formularen. In den Buchbeispielen finden Sie eine kleine MDI-Demo-Anwendung.

7.4.3 SDI-Formulare erzeugen, öffnen und schließen

Formulare erzeugen

Um Formulare zu öffnen, müssen Sie diese zunächst – wie jeden Referenztyp – erzeugen. Das folgende Beispiel geht davon aus, dass das Projekt ein Formular mit dem Klassennamen *Demo* enthält:

```
Demo frmDemo = new Demo();
```

Üblicherweise wird die Formularvariable lokal in der Methode deklariert, in der Sie das Formular öffnen wollen. Dann erzeugen Sie das Formular ebenfalls in dieser Methode.

Wenn Sie daran denken, die Formularvariable auf der Klassenebene zu deklarieren und das Formular direkt dort oder im Konstruktor des Startformulars zu erzeugen (um dieses in Methoden u. U. schneller zu öffnen), sollten Sie eine Tatsache beachten: Wenn der Anwender das Formular über den Schließen-Schalter oder das Systemmenüfeld der Titelleiste oder über Alt F4 *schließt, ruft das Formular seine Dispose-Methode auf. Das führt dazu, dass das Formular seine Ressourcen und sich selbst freigibt (damit der Garbage Collector das Formular zerstören kann). Sie können das Formular dann also nicht ein zweites Mal öffnen, ohne dieses neu zu erzeugen.*

Um Probleme zu vermeiden, sollten Sie Formulare also immer erst erzeugen und dann öffnen.

Formulare öffnen

Ein Formular können Sie entweder normal oder als Dialog öffnen. Wenn Sie ein Formular normal öffnen, kann der Anwender zwischen diesem Formular und den anderen Formularen Ihrer Anwendung wechseln. Normal öffnen Sie ein Formular über die Show-Methode:

```
frmDemo.Show();
```

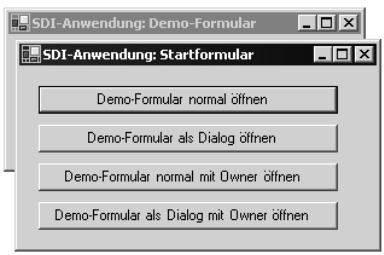

Bild 7.11: Ein Formular wurde normal geöffnet. Der Anwender wechselte danach wieder zum Startformular.

Solche Formulare, die auch als unmodale Formulare bezeichnet werden, kennen Sie bereits von vielen Anwendungen: Die Fenster von Visual Studio sind beispielsweise alle unmodal. Es wäre ja auch dumm, wenn Sie das Eigenschaftenfenster öffnen, und Sie dann nicht mehr zu einem anderen Fenster wechseln können. Üblicherweise werden unmodale Formulare in MDI-Anwendungen eingesetzt, weil sie da am meisten Sinn machen.

Bei unmodalen Formularen müssen Sie beachten, dass Anweisungen, die dem Aufruf der `Show`-Methode folgen, ausgeführt werden, während das Formular geöffnet und angezeigt wird. Das kann in einigen Fällen unerwünschte Nebenwirkungen haben.

Wenn Sie das Formular als Dialog aufrufen wollen, verwenden Sie die `ShowDialog`-Methode:

```
frmDemo.ShowDialog();
```

Nun muss der Anwender das geöffnete Formular erst schließen, damit das darunter liegende Programm weiter ausgeführt wird. Der Anwender kann zudem nicht zwischen dem geöffneten Formular und anderen Formularen der Anwendung wechseln.

Bild 7.12: Ein Formular wurde als Dialog geöffnet. Das Formular bleibt im Vordergrund. Der Anwender kann nicht zum Startformular wechseln.

Dialoge, die auch als modale Formulare bezeichnet werden, sind Ihnen ebenfalls in Form des Datei-Öffnen- oder des Optionen-Dialogs einer beliebigen Anwendung bekannt. Dialoge besitzen für Programmierer den großen Vorteil, dass der Anwender das Formular erst schließen muss, bevor das Programm weiterläuft. Unerwünschte Nebenwirkungen sind damit fast ausgeschlossen.

Parent und Owner

Der Parent (Elter[1]) eines Steuerelements ist der Container auf dem das Steuerelement angelegt ist. Bei einer MDI-Anwendung ist das MDI-Parent-Formular der Parent von MDI-Child-Formularen. Bei SDI-Anwendungen gibt es keinen Parent. Trotzdem können Sie auch bei SDI-Anwendungen in der Eigenschaft `StartPosition` den Wert `CenterParent` einstellen. Das Formular sollte dann so geöffnet werden, dass es auf dem Parent zentriert wird. Obwohl SDI-Formulare gar keinen Parent besitzen, funktioniert das Zentrieren dann, wenn das Formular als Dialog geöffnet wird.

Der Owner (Besitzer) eines Formulars besitzt eine andere Bedeutung. Sie können ein Formular als Owner des anderen angeben. Wird der Owner geschlossen oder verkleinert, wird auch das andere Formular geschlossen bzw. verkleinert. Ein Formular, das einem anderen »gehört«, wird außerdem immer über dem Owner-Formular angezeigt, auch dann, wenn es im Moment nicht aktiviert ist.

1. Einzahl von Eltern

Den Owner definieren Sie bei unmodal geöffneten Formularen vor dem Öffnen, indem Sie das zu öffnende Formular der `AddOwnedForm`-Auflistung des Owners hinzufügen:

```
Demo frmDemo = new Demo();
this.AddOwnedForm(frmDemo);
frmDemo.Show();
```

Bei modal geöffneten Dialogen gibt es normalerweise keinen Owner. Sie können aber einen Owner im ersten Argument der `ShowDialog`-Methode übergeben. Sinn macht das wohl nur in seltenen Fällen: Wenn Sie in einer Methode des Formulars *x* ein Formular *y* als Dialog öffnen und dabei das Formular *z* als Owner angeben, wird das Formular *y* über dem Formular *z* angezeigt. Das Formular *z* wird dazu, wenn notwendig, aus dem Hintergrund hervorgeholt. Ich habe so etwas noch nie benötigt ...

Schließen eines Formulars

Wenn Sie ein Formular schließen wollen, können Sie dazu die `Hide`- und die `Close`-Methode verwenden. `Hide` schließt das Formular, entfernt dieses aber nicht aus dem Arbeitsspeicher. Über die Formularvariable können Sie auf alle Daten des Formulars zugreifen. `Close` schließt das Formular und gibt alle Ressourcen frei. Das Formular wird dann vom Garbage Collector zerstört. Sie können also nicht mehr auf die Eingaben zugreifen.

Eine andere Möglichkeit, ein Formular zu schließen, ist der Eigenschaft `DialogResult` einen anderen Wert als `DialogResult.None` zuzuweisen. Das Formular wird dann nur versteckt. Den Umgang damit beschreibe ich im nächsten Abschnitt.

Wenn der Anwender ein Formular über das Systemmenüfeld, den Schließen-Schalter oder über Alt F4 schließt, wird:

- bei einem modal geöffneten Formular implizit die `Hide`-Methode aufgerufen,
- bei einem unmodal geöffneten Formular implizit die `Close`-Methode aufgerufen.

Modal geöffnete Formulare bleiben dann also im Arbeitsspeicher geladen, wenn der Anwender das Formular über eine der Windows-

Techniken schließt oder wenn Sie `DialogResult` auf einen Wert ungleich `None` setzen. Das dürfte aber kein Problem zu sein. Wenn die Formularvariable ungültig wird, sollte das Formular vom Garbage Collector entsorgt werden. In meinen Tests machte es für den Arbeitsspeicher keinen Unterschied, ob modal geöffnete Formulare über `Close` oder eine der Windows-Techniken geschlossen wurden.

Formulare als Eingabedialog verwenden

Häufig benötigen Sie in einem (Teil-)Programm Eingaben des Anwenders. Eine geschützte Anwendung erfordert beispielsweise beim Start die Eingabe eines Passworts. Für diese Eingaben können Sie dann ein Formular verwenden, das Sie als Dialog (also modal) öffnen. Wenn Sie die Eingaben aber im aufrufenden Programm auswerten wollen, müssen Sie wissen, welchen Button der Anwender betätigt hat, wenn Sie mehrere Buttons zum Schließen des Formulars anbieten.

Um Ihnen die Arbeit mit Dialogen zu erläutern, verwende ich das Beispiel eines Login-Dialogs, der beim Starten einer Anwendung modal geöffnet werden soll, damit der Anwender seinen Benutzernamen und sein Passwort eingeben kann. Dieser Dialog soll einen OK- und einen Abbrechen-Schalter besitzen.

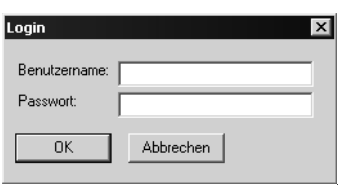

Bild 7.13: Ein Logindialog

Um dem Anwender zu ermöglichen, den OK-Button immer mit ↵ betätigen zu können, sollte dieser Button in die Eigenschaft AcceptButton des Formulars eingetragen werden. Ähnliches gilt für den Abbrechen-Button. Tragen Sie diesen in die Eigenschaft `CancelButton` ein, damit der Anwender auch die Taste Esc betätigen kann, um den Button zu betätigen.

Im Beispielformular aus Abbildung 7.13 habe ich zudem den Minimieren- und den Maximieren-Schalter abgeschaltet (über die Eigenschaften `MinimizeBox` und `MaximizeBox`) und den Rahmen in der Eigenschaft `FormBorderStyle` auf `FixedDialog` gesetzt.

Der Logindialog soll beim Start der Anwendung geöffnet werden. Sie können das Formular also einfach in der Methode `Main` öffnen, bevor die eigentliche Anwendung startet. Alternativ können Sie auch das `Load`-Ereignis des Startformulars verwenden.

Um dort herauszufinden, welcher Schalter im Logindialog betätigt wurde, können Sie einfach die Rückgabe der `ShowDialog`-Methode auswerten. `ShowDialog` gibt den Wert zurück, der zuletzt in der Eigenschaft `DialogResult` des Formulars gespeichert war. Diese Eigenschaft besitzt den Typ `DialogResult`. Dieser Typ ist eine Auflistung mit den Werten `None`, `OK`, `Cancel`, `Abort`, `Retry`, `Ignore`, `Yes` und `No`, die jeweils für einen entsprechenden Button stehen.

Sie können nun in den Ereignisbehandlungsmethoden der Buttons auf dem Loginformular einen entsprechenden Wert in die Eigenschaft `DialogResult` schreiben:

```
private void btnCancel_Click(object sender,
   System.EventArgs e)
{
   this.DialogResult = DialogResult.Cancel;
}
```

Das Formular wird automatisch geschlossen, wenn Sie einen Wert ungleich `DialogResult.None` in diese Eigenschaft schreiben.

Alternativ können Sie schon im Entwurf einen passenden Wert in die gleichnamige Eigenschaft der Buttons schreiben. Dann müssen Sie noch nicht einmal die Ereignisbehandlungsmethode programmieren. Wenn der Anwender den Button betätigt, wird das Formular automatisch geschlossen. Alternativ können Sie bei Buttons mit `DialogResult`-Wert aber auch die Ereignisbehandlungsmethode programmieren, beispielsweise um die Eingaben zu überprüfen. Setzen Sie einfach `DialogResult` auf `DialogResult.None`, wenn Sie das Schließen verhindern wollen.

Wenn der Anwender das Formular über das Systemmenüfeld, den Schließen-Schalter der Titelleiste oder über Alt F4 schließt, wird übrigens der Wert `DialogResult.Cancel` in die `DialogResult`-Eigenschaft geschrieben.

Im Logindialog habe ich mir die Arbeit vereinfacht und den Buttons bereits einen `DialogResult`-Wert vergeben. Der OK-Schalter soll lediglich überprüfen, ob der Benutzername eingegeben ist.

Ein Problem ist dabei noch zu lösen: Die `Main`-Methode soll die Eingaben des Login-Dialogs auswerten. Um dieses Problem zu lösen, können Sie die Textboxen einfach `public` deklarieren (per Voreinstellung sind diese `private`). Im Entwurf können Sie dies über die Eigenschaft `Modifiers` erreichen. Ich gehe aber einen anderen Weg, weil ich denke, Steuerelemente sollten besser privat bleiben, damit das Formular möglichst gut seine Daten kapselt: Deklarieren Sie für die auszuwertenden Eingaben neue `public`-Eigenschaften im Formular:

```
public class LoginForm : System.Windows.Forms.Form
{
   /* Neue Eigenschaften für die
      * Rückgabe der Eingaben */
   public string UserName = "";
   public string Password = "";
   ...
```

In der Ereignisbehandlungsmethode für das `Click`-Ereignis des OK-Buttons werden diese Eigenschaften dann definiert:

```
private void btnOK_Click(object sender,
   System.EventArgs e)
{
   if (txtUserName.Text != "")
   {
      /* Eigenschaften mit den Eingaben
         * belegen */
      this.UserName = txtUserName.Text;
      this.Password = txtPassword.Text;
      /* Geschlossen wird das Formular automatisch,
       * da die DialogResult-Eigenschaft des Button
       * auf DialogResult.OK gesetzt ist */
```

```
    }
    else
    {
        MessageBox.Show("Geben Sie einen " +
            "Benutzernamen ein.");
        /* Schließen verhindern */
        this.DialogResult = DialogResult.None;
    }
}
```

Beim Öffnen des Loginformulars in der `Main`-Methode der Startformular-Klasse fragen Sie den `DialogResult`-Wert ab. Ist dieser `OK`, wird noch der Benutzername und das Passwort überprüft. Bei `Cancel` oder einem ungültigen Benutzernamen bzw. Passwort wird die Anwendung nicht gestartet:

```
[STAThread]
static void Main()
{
    /* Loginformular anzeigen */
    LoginForm frmLogin = new LoginForm();
    if (frmLogin.ShowDialog() == DialogResult.OK)
    {
        /* Benutzername und Passwort überprüfen */
        if (frmLogin.UserName == "Zaphod" &&
            frmLogin.Password == "42")
        {
            /* Anwendung starten */
            Application.Run(new StartForm());
        }
    }
}
```

7.4.4 Formulare initialisieren

Häufig müssen Formulare vor dem Öffnen initialisiert werden. Eigentlich bedarf die Lösung dieses »Problems« gar keinen eigenen Abschnitt im Buch. Dafür gibt es schließlich Konstruktoren. Früher war das Initialisieren aber ein echtes Problem (besonders unter Visual Basic 6). Deshalb will ich ganz kurz erläutern, wie Sie Formulare initialisieren.

Wenn ein Formular sich selbst initialisieren soll, können Sie dazu einfach den Default-Konstruktor verwenden. Achten Sie aber darauf, dass Sie die Initialisierungen nach dem Aufruf von *InitializeComponent* vornehmen. Dann können Sie auf alle Steuerelemente des Formulars zugreifen.

Das folgende Beispiel schreibt den Windows-Loginnamen des Anwenders in die Benutzername-Textbox des Loginformulars:

```
public LoginForm()
{
   InitializeComponent();

   /* Initialisieren des Formulars */
   txtUserName.Text = System.Environment.UserName;
}
```

Wenn das Formular von außen initialisiert werden muss, deklarieren Sie einfach einen neuen Konstruktor. Beim Loginformular wäre es beispielsweise sinnvoll, dass einem Konstruktor ein Benutzername übergeben werden kann. Achten Sie dann aber darauf, dass *InitializeComponent* aufgerufen wird:

```
public LoginForm(string userName)
{
   InitializeComponent();

   /* Initialisieren des Formulars */
   txtUserName.Text = userName;
}
```

Das Loginformular kann nun mit dem Default-Konstruktor erzeugt werden (der den Benutzernamen automatisch ermittelt):

```
LoginForm frmLogin = new LoginForm();
```

oder mit dem zweiten Konstruktor unter Übergabe eines Benutzernamens:

```
LoginForm frmLogin = new LoginForm("Zaphod");
```

7.4.5 Menüs

Normale Menüs

Sie können einem Formular über die `MainMenu`-Komponente auf sehr einfache Weise ein Menü hinzufügen. Ziehen Sie das Steuerelement auf ein Formular. Der Formular-Designer ermöglich die Erstellung des Menüs über die Tastatur.

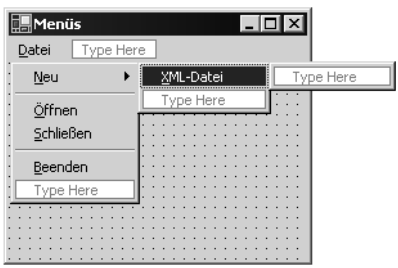

Bild 7.14: Ein Formular mit einem MainMenu-Steuerelement im Entwurf

Geben Sie einfach die Menüpunkte an der Stelle ein, an der Sie diese benötigen. Eine Zugriffstaste können Sie – wie in C# üblich – über ein & vor einem Zeichen definieren. Einen Trennstrich erzeugen Sie, indem Sie einen einfachen Bindestrich eingeben. Navigieren können Sie über die Cursortasten. Bereits eingegebene Menüpunkte können Sie über ↵ wieder zum Editieren öffnen. Einzelne Menüpunkte können Sie mit Hilfe der Maus verschieben oder mit Entf löschen.

Das Menü ist eine Komponente, die in die Eigenschaft `Menu` des Formulars eingetragen ist. Sie können mehrere Menüs erzeugen und diese in der Laufzeit dynamisch zuweisen.

Über das Eigenschaftenfenster können Sie verschiedene Eigenschaften der Menüpunkte einstellen. Die wichtigste ist wohl der Name des Menüpunkts. Ich beginne den Namen immer mit »mnu«. Die anderen Eigenschaften listet Tabelle 7.8 auf.

Eigenschaft	Beschreibung
Checked	definiert, ob der Menüeintrag mit einem Häkchen oder, wenn RadioCkecked true ist, mit einem Punkt im Menü erscheint. Diese Eigenschaft, die im Menü anzeigen soll, dass irgendetwas eingeschaltet ist, wird oft erst in der Laufzeit gesetzt.
DefaultItem	Über diese Eigenschaft können Sie in einem Untermenü einen Defaulteintrag festlegen. Klickt der Anwender doppelt auf dem Untermenü-Eintrag, wird automatisch das Default-Menü betätigt.
Enabled	definiert, ob der Menüeintrag ausgewählt werden kann oder nicht. Diese Eigenschaft wird oft erst in der Laufzeit gesetzt.
MdiList	Diese Eigenschaft gilt nur für MDI-Anwendungen. Wenn Sie diese Eigenschaft einschalten, enthält das Menü in der Laufzeit eine Liste der geöffneten MDI-Childformulare. Der Anwender kann über diese Liste auf ein bestimmtes Formular umschalten.
MergeOrder	Menüs können miteinander kombiniert werden. In einer MDI-Anwendung kann das Menü eines MDI-Childformulars beispielsweise das Menü des MDI-Parentformulars erweitern, wenn das MDI-Childformular aktiviert wird. Über MergeOrder geben Sie die Position des Menüpunkts im kombinierten Menü an.
MergeType	Diese Eigenschaft bestimmt, ob und wie ein Menüpunkt in ein anderes Menü eingefügt wird, wenn zwei Menüs miteinander kombiniert werden.
OwnerDraw	Wenn Sie hier true einstellen, können Sie den Menüeintrag im Ereignis DrawItem selbst zeichnen. Im Ereignis MeasureItem müssen Sie dann die Größe des Menüeintrags definieren.
RadioCheck	definiert, ob das Menü einen Punkt statt einem Häkchen anzeigt, wenn die Eigenschaft Checked true ist
Shortcut	definiert eine optionale Kurztasten-Kombination, über die der Menübefehl jederzeit im Programm aufgerufen werden kann, ohne das Menü öffnen zu müssen

Eigenschaft	Beschreibung
`ShowShortcut`	definiert, ob die in `Shortcut` angegebene Tastenkombination im Menü angezeigt wird
`Text`	definiert den Text des Menüeintrags
`Visible`	definiert, ob der Menüeintrag sichtbar ist. Diese Eigenschaft wird ebenfalls oft erst in der Laufzeit gesetzt.

Tabelle 7.8: Die Eigenschaften eines Menüpunkts

Um auf die Betätigung eines Menüeintrags zu reagieren, werten Sie das `Click`-Ereignis aus.

Menüs können zudem noch dynamisch im Programm erzeugt werden. Das Menü stellt Ihnen dazu eine Auflistung der Menüpunkte in der Eigenschaft `MenuItems` zur Verfügung. Das müssen Sie aber nun selbst ausprobieren ...

Kontextmenüs

Kontextmenüs werden ähnlich erzeugt wie normale Menüs: Ziehen Sie eine `ContextMenu`-Komponente auf das Formular, aktivieren Sie diese (falls dies nicht der Fall ist) und erzeugen Sie die Menüeinträge. Kontextmenüs können nur eine Ebene besitzen. Das Kontextmenü geben Sie dann in der Eigenschaft `ContextMenu` der Steuerelemente an, die dieses besitzen sollen. Fertig.

Beachten Sie, dass viele Steuerelemente bereits vordefinierte Kontextmenüs besitzen, die Einträge für die wichtigsten Aktionen beinhalten.

8 Datenbindung und Datenzugriff

Einige Steuerelemente können Sie mit nur wenigen Zeilen Quellcode an eine Datenquelle binden. Diese Steuerelemente zeigen die Daten der Quelle dann automatisch an. Datenquellen können die verschiedenen Auflistungen, aber auch spezielle ADO.NET-Datenzugriffsobjekte sein. Dieses Kapitel beschreibt zunächst die Prinzipien der Datenbindung und geht danach auf das Abfragen und Bearbeiten von externen Datenquellen mit ADO.NET ein.

8.1 Datenbindung

Viele Steuerelemente können Sie über deren Eigenschaft `DataSource` an eine Datenquelle binden, die von diesen Steuerelementen dann automatisch dargestellt wird. Eine Datenquelle ist im Prinzip dabei nur ein Objekt, das die Schnittstelle `IList` oder `IListSource` (deren `GetList`-Methode eine Referenz auf eine `IList`-Schnittstelle zurückgibt) implementiert. Das Steuerelement verwendet diese Schnittstelle, um die Daten auszulesen und darzustellen.

Die Datenbindung in Windows-Anwendungen unterscheidet sich etwas von der Datenbindung in Webanwendungen (ASP.NET). Datenbindung in Webanwendungen arbeitet auch mit Objekten, die lediglich die Schnittstelle `ICollection` *implementieren. Diese Art von Datenbindung behandle ich in meinem Internetprogrammierungsbuch.*

`IList` definiert die grundlegenden Methoden und Eigenschaften zum sequenziellen Durchgehen von gespeicherten Daten und den Zugriff auf die »Datensätze« über einen Integer-Index. Wenn Sie eine Datenquelle, die `IList` implementiert, an ein Steuerelement binden, verwaltet das Formular die aktuelle Position in dieser Datenquelle. Wird die Position geändert, zeigt das Steuerelement die Daten des Listenelementes an dieser Position an. Über die Eigenschaft `BindingContext` des Formulars können Sie die Position in der Datenquelle beeinflussen.

8.1.1 Das Beispiel

Ich erläutere die Datenbindung an einem praxisnahen Beispiel. Eine `ArrayList`-Auflistung soll mehrere Personen verwalten. Ein Formular soll die Daten der gespeicherten Personen anzeigen.

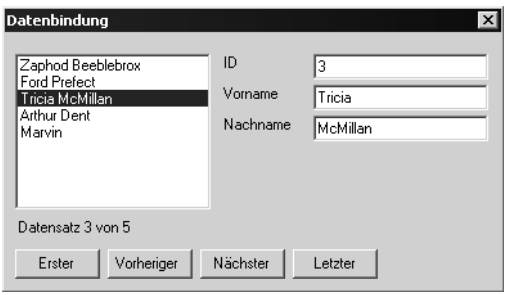

Bild 8.1: Ein Formular, das die in einer Auflistung gespeicherten Personendaten anzeigt

Eine Listbox soll alle Personen auflisten, die im `ArrayList`-Objekt verwaltet werden. Textboxen zeigen die Daten der einzelnen Person an. Über die Schalter unten soll eine Navigation möglich sein.

Die `ArrayList`-Klasse implementiert `IList` und kann deswegen an Steuerelemente gebunden werden. Alternativ könnten Sie auch eine eigene Auflistung für die Personen erzeugen, die Sie von `CollectionBase` ableiten. Ich will das Beispiel aber möglichst einfach halten. Die Klasse zur Speicherung der Personendaten sieht folgendermaßen aus:

```
public class Person
{
   private int id;
   private string firstName;
   private string lastName;

   /* Der Konstruktor */
   public Person(int id, string firstName,
      string lastName)
   {
      this.id = id;
```

```
        this.firstName = firstName;
        this.lastName = lastName;
    }

    /* Eigenschaften mit Zugriffsmethoden */
    public int ID
    {
        get {return id;}
        set {this.ID = value;}
    }

    public string FirstName
    {
        get {return firstName;}
        set {this.firstName = value;}
    }

    public string LastName
    {
        get {return lastName;}
        set {this.lastName = value;}
    }

    /* Die ToString-Methode */
    public override string ToString()
    {
        return this.firstName + " " + this.lastName;
    }
}
```

Wichtig ist, dass Sie zumindest die Eigenschaften, die in Steuerelementen angezeigt werden sollen, mit `get`/`set`-Eigenschaften deklarieren. Das Formular benötigt die `get`-Methode, um den Wert der Eigenschaft auszulesen. Daneben ist die `ToString`-Methode wichtig. Steuerelemente wie die Listbox, die mehrere Datensätze anzeigen, rufen zur Anzeige diese Methode auf.

8.1.2 Einfache Datenbindung

Einfache Datenbindung bedeutet, dass Sie Steuerelemente, die nur einen Wert darstellen können, an eine Datenquelle binden. Die Textboxen in Abbildung 8.1 sind ein Beispiel dafür. Der Typ des Steuerelements muss natürlich zum Typ des Feldes[1] passen. An eine Checkbox werden Sie beispielsweise wohl kaum ein `string`-Feld anbinden können.

Das Steuerelement binden Sie dazu über dessen `DataBindings`-Methode an die Datenquelle an. Die Variable zur Verwaltung der Daten sollte auf der Klassenebene privat deklariert sein. Sie benötigen den Zugriff auf diese Variable, wenn Sie die Position in der Auflistung nachträglich verändern wollen. Der folgende Quelltext zeigt einen Ausschnitt aus der Formular-Klasse:

```
/* private Eigenschaft für die Auflistung */
private ArrayList persons = new ArrayList();

/* Der Konstruktor */
public StartForm()
{
   InitializeComponent();

   /* ArrayList erzeugen und mit Personen füllen */
   persons = new ArrayList();
   persons.Add(new Person(1, "Zaphod", "Beeblebrox"));
   persons.Add(new Person(2, "Ford", "Prefect"));
   persons.Add(new Person(3, "Tricia", "McMillan"));
   persons.Add(new Person(4, "Arthur", "Dent"));
   persons.Add(new Person(5, "Marvin", ""));

   /* Die Auflistung an die Eigenschaft
    * Text der Textboxen anbinden */
   txtID.DataBindings.Add("Text", persons, "ID");
```

1. Ich verwende hier den allgemeinen Begriff »Feld« oder »Datenfeld« an Stelle von »Eigenschaft«, weil Datenbindung auch mit ADO.NET-Datenquellen funktioniert. Und das sind häufig Datenbanken, deren Datensätze eben auf Feldern bestehen.

```
txtFirstName.DataBindings.Add("Text", persons,
   "FirstName");
txtLastName.DataBindings.Add("Text", persons,
  "LastName");
```

Im Konstruktor werden die Steuerelemente an die Personenauflistung gebunden. Im ersten Argument von `DataBindings.Add` geben Sie dazu die Eigenschaft des Steuerelements an, die mit dem Wert des Datenfeldes beschrieben werden soll, das zweite Argument ist das Objekt, das die Daten verwaltet. Im dritten Argument geben Sie den Namen des Datenfeldes an (in unserem Fall den Namen der Eigenschaft des *Person*-Objekts).

Sie können über die `Add`-Methode mehrere Bindungen hinzufügen, damit Sie in Sonderfällen möglichst flexibel programmieren können. Manchmal mag es sinnvoll sein, eine Eigenschaft des Steuerelements an ein Datenfeld einer Datenquelle zu binden und eine andere Eigenschaft an ein anderes Datenfeld, vielleicht sogar einer anderen Datenquelle. Das überlasse ich aber dann ganz Ihnen ...

Das Beispiel zeigt bisher nur den ersten Datensatz an. Um andere Datensätze anzeigen zu können, müssen Sie dem Anwender die Möglichkeit geben, die Position zu wechseln. Dazu können Sie beispielsweise eine Listbox verwenden, die Sie ebenfalls, über die komplexe Datenbindung anbinden.

8.1.3 Komplexe Datenbindung

Komplexe Datenbindung bedeutet, dass ein Steuerelement gleich mehrere Datensätze anzeigen kann. Listboxen und Comboboxen sind Beispiele dafür. Der aktuelle Datensatz wird dabei im Steuerelement auch als der aktuelle angezeigt. Komplexe Datenbindung ist einfach. Dazu binden Sie die gesamte Auflistung an die `DataSource`-Eigenschaft des Steuerelements an:

```
lbxDemo.DataSource = persons;
```

Fertig. Schon zeigt die Listbox unseres Beispiels alle Personen an. Wenn Sie in der Listbox einen anderen Eintrag anklicken, zeigen die Textboxen (leider etwas zeitverzögert) die Daten der aktuellen Person an.

8.1.4 Verändern der Position

Die Schalter des Beispielformulars sollen den aktuellen Datensatz auf eine andere Position setzen. Die Realisierung ist einfach: Über die `BindingContext`-Eigenschaft des Formulars erreichen Sie das Bindungskontext-Objekt für eine Auflistung. Ein Formular kann mehrere Bindungskontexte verwalten. `BindingContext` ist deswegen eine Auflistung, die als Schlüssel ein beliebiges Objekt verwaltet. Die Auflistung geben Sie deswegen als Schlüssel an. Über die Eigenschaft `Position` können Sie die Position beeinflussen. Der folgende Auszug aus dem Formularquelltext zeigt die Ereignisbehandlungsmethoden für die Buttons:

```
private void btnFirst_Click(object sender,
   System.EventArgs e)
{
   /* Auf den ersten Datensatz stellen */
   this.BindingContext[persons].Position = 0;
}

private void btnPrevious_Click(object sender,
   System.EventArgs e)
{
   /* Auf den vorherigen Datensatz stellen,
    * wenn die Auflistung nicht schon auf dem
    * ersten Datensatz steht */
   if (this.BindingContext[persons].Position > 0)
      this.BindingContext[persons].Position--;
}

private void btnNext_Click(object sender,
   System.EventArgs e)
{
   /* Auf den nächsten Datensatz stellen,
    * wenn die Auflistung nicht schon auf dem
    * letzten Datensatz steht */
   if (this.BindingContext[persons].Position <
      this.BindingContext[persons].Count)
      this.BindingContext[persons].Position++;
```

```
}

private void btnLast_Click(object sender,
   System.EventArgs e)
{
   /* Auf den letzten Datensatz stellen */
   this.BindingContext[persons].Position =
      this.BindingContext[persons].Count - 1;
}
```

Achten Sie darauf, dass Sie die Position nicht auf einen Wert kleiner o oder größer `Count - 1` einstellen.

Das Bindungskontext-Objekt besitzt das Ereignis `PositionChanged`, das aufgerufen wird, wenn der aktuelle Datensatz wechselt. Wenn Sie dies auswerten wollen, schreiben Sie (von Hand) eine Ereignisbehandlungsmethode:

```
private void persons_PositionChanged(object sender,
   System.EventArgs e)
{
   lblInfo.Text = "Datensatz " +
      this.BindingContext[persons].Position +
      " von " + this.BindingContext[persons].Count;
}
```

Das Beispiel schreibt eine Information über die aktuelle Position in ein Label.

Die Ereignisbehandlungsmethode müssen Sie dann im Konstruktor des Formulars dem Ereignis zuweisen:

```
public StartForm()
{
   ...
   this.BindingContext[persons].PositionChanged +=
      new System.EventHandler(persons_PositionChanged);
}
```

Und schon ist das Beispiel fertig. Den kompletten Quelltext finden Sie natürlich in den Buchbeispielen.

8.1.5 Automatisches Editieren

Ein Feature der Datenbindung ist, dass Änderungen, die der Anwender an den Daten vornimmt, bei einem Wechsel der Position in die Datenquelle zurückgeschrieben werden. Probieren Sie das einmal aus. So können Sie dem Anwender eine einfache Möglichkeit geben, Daten zu editieren. Sinnvoll ist das Ganze natürlich nur, wenn Sie die Daten beim Öffnen des Formulars aus einer (XML-)Datei oder Datenbank auslesen und beim Schließen des Formulars wieder wegschreiben. Wenn Sie das Editieren verhindern wollen, sperren Sie die Steuerelemente einfach über die `ReadOnly`-Eigenschaft.

Wenn Sie die Daten in unserem Beispiel editieren, zeigt die Listbox nicht die geänderten Daten an. Das ist auch logisch, denn die Listbox wird nur einmal gefüllt und fragt die Daten danach selbst nicht mehr ab. Die Lösung für dieses Problem habe ich noch nicht gefunden.

Zur Verwaltung des Editierens stellt Ihnen das Bindungskontext-Objekt einige Methoden zur Verfügung.

Methode	Bedeutung
`void AddNew()`	erzeugt einen neuen Datensatz. Die Datenquelle muss diese Methode aber unterstützen, was bei einem ArrayList-Objekt beispielsweise nicht der Fall ist.
`void CancelCurrentEdit()`	bricht den aktuellen Editiervorgang ab. Die Änderungen des Benutzers werden verworfen. Die Datenquelle muss diese Methode unterstützen.
`void EndCurrentEdit()`	beendet den aktuellen Editiervorgang. Die Änderungen werden dabei in den aktuellen Datensatz übernommen. Die Datenquelle muss diese Methode unterstützen. Diese Methode sollten Sie beispielsweise im `Closing`-Ereignis des Formulars aufrufen, damit auch die letzten Änderungen übernommen werden.

Methode	Bedeutung
`void RemoveAt(` `int index)`	löscht den Datensatz am angegebenen Index. Die Datenquelle muss das Löschen unterstützen.

Tabelle 8.1: Die Editiermethoden des Bindungskontext-Objekts

8.1.6 Weitere Features, die hier nicht besprochen werden

Datenbindung mit dem DataGrid-Steuerelement

Das DataGrid-Steuerelement eignet sich sehr gut, um Datensätze mit mehreren Feldern anzuzeigen. Sie binden eine Datenquelle wieder ganz einfach über die `DataSource`-Eigenschaft an. Mit dem DataGrid können Sie aber eine Menge andere Dinge anstellen. So können Sie in einer Spalte beispielsweise eine Combobox darstellen, die mit einer Liste von Werten gefüllt ist. Das DataGrid zeigt dann immer den passenden Eintrag aus der Combobox an und der Benutzer kann den Wert über die Combobox ändern.

Master-/Detailformulare

Der Bindungskontext eines Formulars kann nicht nur die aktuelle Position einer Datenquelle, sondern auch deren Beziehungen zu anderen Datenquellen verwalten. So ist es auf einfache Weise möglich, Detaildaten zu einem Datensatz anzuzeigen. Einzelne Steuerelemente können beispielsweise die Daten eines Kunden anzeigen, ein DataGrid zeigt die Bestellungen des aktuellen Kunden an.

8.2 Datenzugriff mit ADO.NET

Über die Klassen der ADO.NET-Bibliothek, die Sie im Namensraum `System.Data` und einigen untergeordneten Namensräumen finden, können Sie beliebige Datenquellen auslesen, darstellen und bearbeiten. ADO.NET sieht nicht nur Datenbanken als Datenquelle an, sondern alles, was irgendwie Daten liefern kann. Dazu gehören beispielsweise auch XML-Dateien, E-Mail-Dienste und Such-Server (wie der Microsoft Index Server). Das einzige, was Sie zum Zugriff benötigen, ist ein passender Provider. Bevor ich beschreibe, was ein Provider ist, kläre ich zunächst ein paar wichtige Begriffe.

ADO.NET ist sehr komplex. Ganze (dicke) Bücher behandeln ausschließlich dieses Thema. Ich beschreibe in diesem Buch nur die Grundlagen und die für die Praxis wichtigsten Features.

8.2.1 Einige Infos zuvor

Notwendiges Basiswissen

Ich kann in diesem Buch nicht erläutern was eine Datenbank ist. Sie sollten also die Grundlagen von Datenbanken verstehen.

Ausnahmebehandlung

Gerade bei der Arbeit mit Datenbanken sind Unmengen von Ausnahmen möglich. Wenn Sie beispielsweise versuchen, Verbindung zu einem SQL-Server aufzunehmen, der nicht läuft, generiert der Provider eine Ausnahme. Sie müssen also eigentlich überall eine Ausnahmebehandlung vorsehen. In den Beispielen verzichte ich wegen der besseren Übersicht und aus Platzgründen auf eine Ausnahmebehandlung.

Wenn Sie keine Ausnahmebehandlung vorsehen und Visual Studio bei einer Ausnahme anhält, ist die Ausnahme-Meldung häufig wenig aussagekräftig. Visual Studio meldet häufig nur einen »Systemfehler« ohne weiteren Text. Wenn Sie die Ausnahme dann aber behandeln und die Eigenschaft `Message` *der Ausnahme ausgeben, ist der Text wesentlich aussagekräftiger. Bevor Sie also ohne weitere Informationen stundenlang nach einem eventuellen Fehler suchen, sehen Sie besser eine (eventuell globale) Ausnahmebehandlung vor oder schalten das JIT-Debuggen für diese Anwendung komplett ab, wie ich es in Kapitel 6 beschrieben habe.*

Beispiele in Konsolenanwendungen

Die meisten Beispiele zum Datenbankzugriff sind aus Gründen der Übersichtlichkeit in Konsolenanwendungen verfasst.

Die Beispieldatenbank

Für einige Beispiele dieses Kapitels verwende ich die *Northwind*-Datenbank des SQL-Servers. Diese Datenbank speichert Artikel-, Kunden- und Bestelldaten eines Vertriebs für Lebensmittel. Sie können die Northwind-Datenbank und die dazu notwendige SQL-Server-Instanz über die Quickstart-Tutorials installieren, wie ich es bereits in Kapitel 1 beschrieben habe. Sie installieren dann die MSDE (Microsoft Data Engine), einen vereinfachten, aber in den grundlegenden Dingen voll funktionsfähigen SQL Server. Leider fehlt dabei ein Programm zur Administration. Dazu können Sie aber den Server-Explorer von Visual Studio in eingeschränkter Form verwenden. Sie können damit die Tabellenstruktur erforschen (und auch verändern), die gespeicherten Daten anschauen und diese sogar editieren.

Für den Zugriff auf eine Access-Datenbank verwende ich die Datenbank *NWind.mdb*, die ähnlich aufgebaut ist wie die *Northwind*-Datenbank. Sie finden eine verkleinerte Version dieser Datenbank als separaten Download auf der Internetseite zum Buch.

8.2.2 ADO, OLEDB, ODBC, MDAC und SQL

ADO und OLEDB

ADO (ActiveX Data Objects) ist die alte Datenzugriffstechnologie von Microsoft. Eigentlich müsste ich ADO gar nicht beschreiben, es gibt ja schließlich ADO.NET. Leider existieren für ADO.NET zurzeit noch nicht allzu viele Provider. Den Zugriff auf Access-, Oracle- oder SQL-Server-6.x-Datenbanken können Sie beispielsweise zurzeit nur über ADO oder ODBC programmieren. Das .NET-Framework integriert aber einen ADO.NET-Provider für ADO, über den Sie, mit einigen Einschränkungen, ADO genauso verwenden können, wie ADO.NET.

ADO arbeitet ebenfalls mit Providern. Wenn Sie den .NET-Provider für ADO verwenden, müssen Sie für ADO einen zusätzlichen Provider angeben. Das ist aber eigentlich ganz einfach: Der Provider wird einfach im Verbindungsstring definiert.

ADO selbst und die Provider sind nicht in das .NET-Framework integriert, sondern als separate COM-Komponente auf dem Rechner in-

stalliert. Das COM-Modell will ich hier gar nicht diskutieren. Irgendwann wird COM so veraltet sein, dass keiner mehr daran denkt.

Ein Begriff, der häufig im Zusammenhang mit ADO fällt, ist OLEDB. OLEDB ist die Basistechnologie von ADO.

ODBC

ODBC (Open Database Access) ist eine noch wesentlich ältere Datenzugriffstechnik. ODBC arbeitet mit DLL[2]-Dateien, die den Zugriff auf die Datenbank ermöglichen. ODBC können Sie für den Zugriff auf ältere Datenbanken, für die kein ADO- oder ADO.NET-Provider zur Verfügung steht, nutzen. ODBC direkt zu verwenden ist sehr kompliziert. ADO und ADO.NET bieten aber je einen Provider, über den die Verwendung von ODBC wieder einfach wird.

MDAC

MDAC (Microsoft Data Access Components) ist eine »Komponente«, die die aktuellen ADO-Provider, aktuelle ODBC-Treiber und verschiedene Tools enthält. Um ADO auf Ihrem System zu installieren, müssen Sie MDAC installieren. Aber das haben Sie ja bereits bei der Installation des .NET-Framework oder von Visual Studio erledigt.

SQL

SQL (Structured Query Language) ist eine standardisierte Sprache für die Abfrage und Manipulation von Daten in relationalen Datenbanken. Alle wichtigen Datenbanksysteme kennen SQL, leider meist mit leichten Abweichungen zum Standard. Mit SQL können Sie Daten abfragen, einfügen, ändern und löschen. Daneben können Sie mit SQL die Datenbanken erzeugen und deren Schema definieren (Tabellen erzeugen, verändern, löschen etc.). SQL stellt dazu einige Befehle zur Verfügung. Über den `SELECT`-Befehl können Sie beispielsweise Daten abfragen, der `INSERT`-Befehl ermöglicht das Einfügen von Daten. SQL wird in diesem Buch nicht besprochen. Mein Online-Artikel

2. Für alle die, die erst bei .NET in die Programmierung eingestiegen sind: Eine DLL-Datei (Dynamic Link Library) ist eine kompilierte Datei, die simple Funktionen enthält. Eine DLL-Datei können Sie (auch in C#) dynamisch in Programme einbinden, sodass Sie die enthaltenen Funktionen aufrufen können.

»SQL« behandelt dieses Thema sehr umfassend. SQL ist für die Praxis sehr wichtig. Die meisten Datenbankprobleme können Sie über SQL lösen. ADO.NET und ADO setzen SQL übrigens konsequent im Hintergrund ein. Wenn Sie über ein `DataSet`-Objekt Daten ändern, aktualisieren, anfügen oder löschen, dann erzeugt dieses (über DataAdapter-Objekt, aber das wird noch behandelt) entsprechende SQL-Anweisungen, wenn es sich bei der Datenquelle um eine relationale Datenbank handelt.

8.2.3 Der Provider

Zum Zugriff auf Daten benötigen Sie einen Provider. Ein Provider stellt die Verbindung zwischen Ihrem Programm und der Datenquelle her. Der Hersteller einer Datenquelle kann diesen zur Verfügung stellen. Ein Provider ist eine Assemblierung, die mehrere Klassen enthält und die direkt in das .NET-Framework integriert wird. Das .NET-Framework beinhaltet bereits einen Provider für den SQL Server ab Version 7 und einen Provider für die Verwendung der alten ADO-Komponente. Bei Microsoft können Sie noch einen Provider für ODBC downloaden. Suchen Sie bei *search.microsoft.com/us/dev* nach »ODBC .NET Data Provider« um den Download zu finden (die Download-URL ist ziemlich komplex und wird sich mit Sicherheit ändern).

Sie sollten, wenn möglich, immer den Provider für den direkten Zugriff verwenden. Der SQL-Server-Provider nutzt z. B. direkt das API des SQL Servers zum Zugriff auf die Daten, ein Oracle-Provider wird das API von Oracle direkt nutzen. Gehen Sie stattdessen über den alten ADO-Provider, verwenden Sie eine weitere Zugriffsschicht (die den Zugriff nicht gerade beschleunigt). Außerdem ist ein ADO.NET-Provider wesentlich einfacher aufgebaut als ein ADO-Provider und arbeitet deshalb performanter und ressourcenschonender.

Die einzelnen Klassen der Provider müssen bestimmte Schnittstellen implementieren. Deswegen ist die Basisfunktionalität dieser Klassen immer gleich. Die Klasse, die zur Erzeugung einer Verbindung zur Datenquelle verwendet wird, muss z. B. die Schnittstelle `IDbConnection` implementieren. Die Eigenschaften und Methoden, die in dieser

Schnittstelle beschrieben sind, finden Sie deshalb in allen Klassen zur Erzeugung einer Verbindung. Die Implementierung dieser Eigenschaften und Methoden ist in den verschiedenen Klassen natürlich vollkommen unterschiedlich.

8.2.4 Übersicht über die Klassen

ADO.NET enthält eine Vielzahl an Klassen zum Datenzugriff. Die meisten werden allerdings nur als Datentyp von Eigenschaften der fundamentalen Klassen verwendet. Einige grundlegende ADO.NET-Klassen finden Sie im Namensraum `System.Data`. Diese Klassen sind nicht providerspezifisch, werden aber von den Providern verwendet. Providerspezifische Klassen finden Sie in eigenen Namensräumen für die einzelnen Provider. Der Namensraum `System.Data.SqlClient` enthält die Klassen des SQL-Server-Providers, der Namensraum `System.Data.OleDb` die Klassen des ADO-Providers und der Namensraum `System.Data.ODBC` die Klassen des ODBC-Providers. Ich gehe davon aus, dass Sie diese Namensräume bei Bedarf über eine `using`-Direktive einbinden. Sie erkennen den Provider am Klassennamen, da dieser immer mit *Sql*, *Ole* bzw. *Odbc* beginnt. Die wichtigsten Klassen beschreibe ich erst einmal nur kurz. Die teilweise komplexen Zusammenhänge verstehen Sie wahrscheinlich (wie es bei mir auch der Fall war) erst, wenn Sie mit diesen Klassen arbeiten.

- Connection-Klassen: Über Instanzen dieser providerspezifischen Klassen bauen Sie eine Verbindung zur Datenquelle auf, die die Basis für alle weiteren Aktionen ist.
- Command-Klassen: Diese ebenfalls providerspezifischen Klassen repräsentieren einen Befehl, der gegen die Datenquelle ausgeführt wird. Über ein `SqlCommand`-Objekt können Sie beispielsweise Daten abfragen und über das zurückgegebene `DataReader`-Objekt auslesen. Sie können aber auch Befehle ausführen, die Daten anfügen, ändern oder löschen.
- DataReader-Klassen: Objekte dieser ebenfalls providerspezifischen Klassen werden verwendet, um die von einem Command-Objekt ermittelten Daten sequenziell durchzugehen und auszulesen.
- `DataSet`-Klasse: Diese Klasse ermöglicht das Auslesen und das flexible Editieren von Daten.

- DataAdapter-Klassen: Diese providerspezifischen Klassen verbinden ein `DataSet`-Objekt (über die Verbindung) mit der Datenquelle. Ein DataAdapter-Objekt verwendet vier interne Command-Objekte für das Abfragen, Anfügen, Aktualisieren und Löschen von Datensätzen in der Datenquelle.

Abbildung 8.2 zeigt die Beziehungen zwischen diesen Klassen. Die Pfeile skizzieren den möglichen Datenfluss. Der Pfeil zwischen dem DataReader und dem `DataSet` steht dafür, dass ein DataReader auch ein `DataSet` füllen kann.

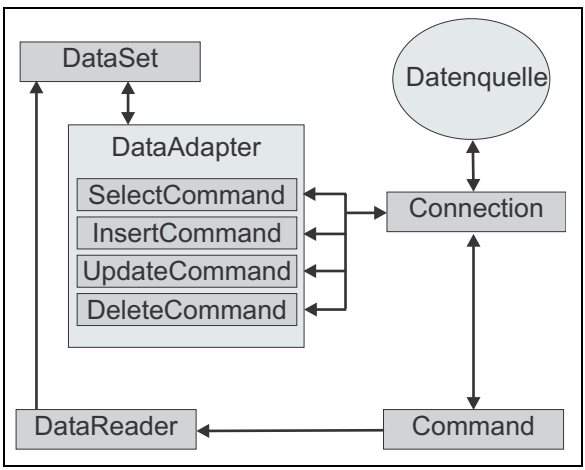

Bild 8.2: Übersicht über die fundamentalen ADO.NET-Klassen

8.2.5 Verbindung zur Datenquelle aufnehmen

Um mit ADO.NET-Datenquellen arbeiten zu können, benötigen Sie immer zuerst eine Verbindung zur Datenquelle. Diese bauen Sie über Instanzen einer Verbindungsklasse auf. Für SQL-Server-Datenbanken verwenden Sie eine Instanz der Klasse `SqlConnection`, für ADO-Datenquellen eine Instanz von `OleDbConnection`, ODBC-Verbindungen werden über die Klasse `OdbcConnection` programmiert. Diese Verbindung geben Sie dann immer an, wenn Sie über die anderen ADO.NET-Klassen Daten abfragen oder manipulieren wollen. Tabelle 8.2 listet die wichtigsten Eigenschaften und Methoden auf.

Eigenschaft / Methode	Beschreibung
`ConnectionString`	verwaltet den Verbindungsstring
`ConnectionTimeout`	definiert das Timeout für den Verbindungsaufbau
`void Open()`	öffnet die Verbindung
`void Close()`	schließt die Verbindung
`SqlTransaction BeginTransaction(...)`	startet eine Transaktion. Transaktionen werden in diesem Buch nicht behandelt.

Tabelle 8.2: Die wichtigsten Eigenschaften und Methoden der Verbindungsklassen

Wo verbinden?

Eine wichtige Frage bei der Datenbankprogrammierung (vor allen Dingen mit Client-/Server-Datenbanken) ist, an welcher Stelle im Programm Sie eine Verbindung aufnehmen und wie lange diese geöffnet bleiben soll. Die Antwort ist ganz einfach: Öffnen Sie die Verbindung da, wo Sie diese benötigen und geben Sie diese so schnell wie möglich wieder frei. Wenn Sie in einer Methode nur kurz auf eine Datenbank zugreifen müssen und die Verbindung danach nicht mehr (oder nicht allzu oft) benötigen, öffnen Sie die Verbindung lokal in der Methode und schließen diese dort auch wieder. Wenn Sie die Verbindung innerhalb einer Klasse (eines Formulars) öfter und/oder mehrfach benötigen, öffnen Sie die Verbindung im Konstruktor der Klasse und schließen diese in einer *Dispose*-Methode (siehe Kapitel 4) bzw. bei einem Formular im `Closing`-Ereignis. Dann müssen Sie die Variable für die Verbindung natürlich auf der Klassenebene (`private`) deklarieren. Öffnen Sie die Verbindung nicht beim Start der Anwendung und lassen Sie diese so lange geöffnet, wie die Anwendung geöffnet ist. Verbindungen kosten auf dem Server Ressourcen. Sie sollten diese Ressourcen nicht unnötig verbrauchen. Außerdem sind viele Datenbankserver so lizenziert, dass eine bestimmte Maximalanzahl von Verbindungen möglich ist. Wenn viele Anwendungen eine Verbindung permanent aufbauen, ist diese Anzahl sehr schnell überschritten. Mit einem nur temporären Verbindungsaufbau können mehr Anwendungen gleichzeitig mit einer Datenbank arbeiten als die

Anzahl der möglichen Verbindungen eigentlich vorgibt. Die ADO.NET-Provider (und die ADO-Provider) unterstützen das bedarfsgerechte Verbinden über das so genannte Connection Pooling: Die Verbindung bleibt im Hintergrund noch einige Zeit geöffnet, auch wenn Sie diese schließen. Benötigen Sie oder ein anderer Client eine Verbindung, wird diese sehr schnell reaktiviert. Außerdem arbeitet das wichtige `DataSet`-Objekt, über das Sie Daten sehr flexibel einlesen und editieren können, sowieso prinzipiell verbindungslos. Es benötigt die Verbindung nicht, während die Daten editiert werden, sondern nur zum Einlesen der Daten und erst dann wieder, wenn *alle* Änderungen in einem »Rutsch« in die Datenbank übertragen werden sollen.

SQL Server

Wenn Sie auf eine SQL Server-Datenbank zugreifen wollen, benötigen Sie zum Verbindungsaufbau ein Objekt der Klasse `System.Data.SqlClient.SqlConnection`. Diese Klasse erwartet im Konstruktor einen Verbindungsstring mit Informationen zu der gewünschten Verbindung. Ein typischer Verbindungsstring für eine Verbindung zur Datenbank *Pubs* auf dem Server *Zaphod* mit der User-ID *sa* und dem Passwort *bandit* sieht so aus:

```
string conString = @"Server=Zaphod;" +
   "Database=Northwind;User ID=sa;Password=Bandit";
```

Der Name des SQL Servers entspricht normalerweise dem Rechnernamen. Wenn Sie zum lokalen SQL Server verbinden wollen, können Sie als Servernamen auch »(local)« eingeben.

Wollen Sie eine Verbindung zu einer lokalen MSDE-Instanz herstellen, müssen Sie den Namen dieser Instanz übergeben. Der Name einer MSDE-Instanz besteht aus dem Rechnernamen, gefolgt von einer speziellen Unterbezeichnung, die bei der Installation durch das .NET-Framework-SDK mit *NetSDK* bezeichnet wird. Auf dem Rechner *Zaphod* sprechen Sie die MSDE-Instanz also über den Namen »Zaphod\NetSDK« an. Um die lokale Instanz anzusprechen, können Sie auch »(local)\NetSDK« verwenden.

Daneben erlaubt die MSDE-Instanz scheinbar keinen Login mit SQL-Server-spezifischen User-Ids, sondern nur mit dem Windows-Benut-

zernamen. Das Argument *Trusted_Connection* muss deswegen auf *yes* gesetzt werden:

```
string conString = @"Server=Zaphod\NetSDK;" +
   "Trusted_Connection=Yes;Database=Northwind";
```

Dann können Sie die Verbindung erzeugen und öffnen:

```
SqlConnection connection = new SqlConnection(conString);

/* Verbindung öffnen */
connection.Open();

/* Einige Infos ausgeben */
Console.WriteLine("Verbindung zum Server " +
   connection.DataSource + ", Version " +
   connection.ServerVersion + " aufgebaut.");

/* Verbindung schließen */
connection.Close();
```

Der Verbindungsstring besitzt eine große Anzahl möglicher Argumente. Ich beschreibe hier nur die wichtigsten. Die einzelnen Argumente werden recht gut in der .NET-Hilfe erläutert.

Verbindungs-argument	Bedeutung
Server	In diesem Argument geben Sie den Namen des SQL Servers an.
Database	spezifiziert die Datenbank
Connection Timeout	definiert das Timeout für den Verbindungsaufbau. Die Voreinstellung beträgt 15 Sekunden. So lange versucht der Provider, eine Verbindung aufzubauen. Läuft das Timeout ab, wird eine Ausnahme erzeugt.

Verbindungs-argument	Bedeutung
Trusted_Connection	spezifiziert, dass eine vertraute Verbindung verwendet werden soll. Wenn Sie hier »yes« eintragen, verwendet der SQL-Server für den Login automatisch die Benutzerinformationen von Windows. Sie müssen dann im Verbindungsstring keine separaten Logininformationen angeben. Der aktuelle Windows-Benutzer muss im SQL-Server als SQL-Server-Benutzer eingetragen sein.
User Id	In diesem Argument übergeben Sie den SQL-Server-Loginnamen des Benutzers an, wenn Sie keine vertraute Verbindung verwenden.
Password	spezifiziert das Passwort, wenn keine vertraute Verbindung verwendet wird

Tabelle 8.3: Die wichtigsten Verbindungsargumente des SQL-Server-Providers

Andere Datenquellen, für die nur ein ADO-Provider verfügbar ist

Wenn Sie eine andere Datenquelle bearbeiten wollen, für die zurzeit noch kein ADO.NET-Provider verfügbar ist, können Sie dazu den ADO.NET-Provider für ADO verwenden. Der Verbindungsaufbau zu einer Datenquelle gestaltet sich dann identisch zu ADO.

Für eine ADO-Verbindung müssen Sie ein Objekt der Klasse `OleDbConnection` erzeugen. ADO verwendet Verbindungsstrings, die denen von ADO.NET ähnlich sind. Sie müssen allerdings im Argument *Provider* den ADO-Provider spezifizieren. Die wichtigsten ADO-Provider sind *Microsoft.Jet.OLEDB.4.0* (der Jet-Provider), *SQLOLEDB* (für den Zugriff auf SQL-Server-6.5-Datenbanken) und *OraOLEDB.Oracle* (für Oracle-Datenbanken). Der Jet-Provider kann alle Datenbanken bearbeiten, die auch Access bearbeiten kann (Access-, DBase-, Foxpro-, Text-, Excel-Datenbanken etc.). Den Oracle-Provider müssen Sie zunächst von Oracle downloaden und installieren.

Access-Datenbanken

Für Access-Datenbanken steht zurzeit nur ein ADO-Provider zur Verfügung. Wenn Sie eine Verbindung zu einer *ungeschützten* Access-Datenbank erzeugen wollen, verwenden Sie den Jet Provider (*Microsoft.Jet.OLEDB.4.0*) und geben im Argument *Data Source* des Verbindungsstrings den vollen Dateinamen der Datenbank an.

Das folgende Beispiel öffnet eine Verbindung zur Datenbank *NWind.mdb*, die im Ordner *C:\Daten* erwartet wird:

```
OleDbConnection connection = new OleDbConnection(
   @"Provider=Microsoft.Jet.OLEDB.4.0;" +
   "Data Source=C:\Daten\NWind.mdb");
connection.Open();
```

Ist die Datenbank mit einem Datenbank-Passwort geschützt, so geben Sie dies im Verbindungsstring-Argument *Jet OLEDB:Database Password* an. Wenn die Datenbank über eine Arbeitsgruppe geschützt ist, müssen Sie die Systemdatenbank angeben, in der die Benutzer eingetragen sind. Diese Angabe erfolgt im Argument *Jet OLEDB:System database*. Vergessen Sie nicht die Angabe des Benutzernamens des Passworts (in den Argumenten *User Id* und *Password*). Das folgende Beispiel öffnet eine Verbindung zu einer Access-Datenbank, die mit Datenbank-Passwort und in einer Arbeitsgruppe geschützt ist:

```
OleDbConnection connection = new OleDbConnection(
   @"Provider=Microsoft.Jet.OLEDB.4.0;" +
   @"Data Source=C:\Daten\NWind.mdb;" +
   "Jet OLEDB:Database Password=Merlin" +
   @"Jet OLEDB:System Database=C:\Daten\System.mdw" +
   "User Id=Arthur;Password=42";
connection.Open();
```

Oracle-Datenbanken

Oracle-Datenbanken bearbeiten Sie idealerweise über den Oracle-ADO-Provider (Microsoft stellt auch einen Provider zur Verfügung, der aber nur eingeschränkt arbeitet). Zurzeit existiert leider noch kein ADO.NET-Provider für Oracle.

Für Oracle-Verbindungen muss, anders als beim SQL Server, auf dem Client eine Verbindungssoftware installiert werden. Für Oracle 8-Datenbanken ist dies zurzeit *Net8*. Net8 wird bei der Installation des Providers mitinstalliert. Den aktuellen Provider finden Sie bei *www.oracle.com*. Die Größe des Downloads beträgt allerdings etwa 40 MB. Klicken Sie auf den Link DOWNLOADS und wählen Sie aus der Utility-/Driver-Liste den ORACLE PROVIDER FOR OLEDB (oder einen .NET-Provider falls verfügbar). Wenn Sie noch nicht im *Oracle Technology Network* registriert sind, müssen Sie sich vor dem Download (kostenlos) registrieren. Wenn Sie den Provider installieren, achten Sie darauf, dass Sie diesen nicht in ein bereits vorhandenes Oracle-Verzeichnis einer älteren Oracle-Version installieren. Das würde mit ziemlicher Sicherheit zu Problemen führen. Installieren Sie neben dem Oracle-Provider auch den Net8-Client, wenn dieser auf Ihrem System noch nicht installiert ist.

Dann müssen Sie eventuell noch mit dem *Net8 Assistant* einen so genannten *Dienst* einrichten. Über diesen Dienst identifiziert Net8 später die zu verwendende Oracle-Instanz. Standardmäßig ist normalerweise der Dienst *ORCL* eingerichtet, der entweder auf die lokale Oracle-Instanz zeigt oder, wenn diese nicht vorhanden ist, auf eine entfernte, bei der Installation von Net8 angegebene Instanz. Sie müssen also im Net8 Assistant eventuell einen neuen Dienst anlegen, der auf die korrekte Oracle-Instanz zeigt. Klicken Sie dazu in diesem Assistenten auf das Plus-Symbol, um einen neuen Dienst anzulegen. Den Rest erledigen Sie zusammen mit dem Assistenten. Den so erstellten Dienst geben Sie dann beim Öffnen der Verbindung im Verbindungsstring im Argument *Data Source* an. Der Provider wird mit *OraOLEDB.Oracle* bezeichnet:

```
OleDbConnection connection = new OleDbConnection(
   "Provider=OraOLEDB.Oracle;" +
   "Data Source=ORCL;" +
   "User ID=Scott;Password=Tiger");
connection.Open();
```

Dieses Beispiel konnte ich mit ADO.NET leider nicht testen, da der Oracle-Provider auf meinem aktuellen System leider bei der Installation diverse Fehler meldete und nach der Installation Oracle nicht mehr lief. Ich hatte mich auch nicht an meinen eigenen Tipp gehalten, den Provider nicht in das Oracle-Verzeichnis zu installieren ... Der Zugriff funktionierte unter dem alten ADO genau wie hier beschrieben.

8.2.6 Die Befehlsklassen

Jeder Provider beinhaltet eine Klasse, über die Befehle gegen die Datenbank ausgeführt werden. Bei den vordefinierten Providern heißen diese Klassen `SqlCommand`, `OleDbCommand` und `OdbcCommand`. Bei relationalen Datenbanken repräsentieren die Instanzen dieser Klassen SQL-Anweisungen (inklusive Befehlen, die gespeicherte Prozeduren ausführen). Sie können selbst eine Instanz einer Befehlsklasse erzeugen, um damit einen Befehl auszuführen. Handelt es sich dabei um eine Abfrage von Daten, erhalten Sie ein `DataReader`-Objekt zurück, über das Sie die Daten lesen können.

Die wichtigsten Eigenschaften und Methoden dieser Klassen zeigt Tabelle 8.2.

Eigenschaft/Methode	Beschreibung
`CommandText`	verwaltet den Befehl
`CommandTimeout`	definiert den Timeout für die Ausführung des Befehls
`CommandType`	definiert die Art es Befehls. Sie können hier die Werte der Auflistung `CommandType` eintragen: `StoredProcedure`, `TableDirect` oder `Text`.
`int ExecuteNonQuery()`	führt einen Befehl aus, der eine Aktion (wie Einfügen, Ändern oder Löschen) in der Datenquelle bewirkt. Gibt die Anzahl der vom Befehl betroffenen Datensätze zurück.

Eigenschaft/Methode	Beschreibung
DataReader ExecuteReader()	führt einen Befehl gegen die Verbindung aus, der Daten abfragt. Gibt ein providerspezifisches DataReader-Objekt zurück.
void Prepare()	sorgt dafür, dass der Befehl von der Datenquelle kompiliert wird, damit eine spätere Ausführung beschleunigt wird.

Tabelle 8.4: Die wichtigsten Eigenschaften und Methoden der Befehlsklassen

8.2.7 Daten über DataReader-Objekte lesen

Die `ExecuteReader`-Methode eines Befehlsobjekts gibt ein Objekt einer DataReader-Klasse zurück `SqlDataReader`, `OleDbDataReader` oder `OdbcDataReader`). Mit Hilfe dieser Objekte können Sie Daten auslesen. Sie können allerdings nur einmal durch die Daten gehen (nicht zurückspringen) und nichts in die Daten hineinschreiben (die Daten sind schreibgeschützt).

Das Prinzip

Das DataReader-Objekt bleibt während des Lesens mit der Datenquelle verbunden, bis es geschlossen wird. Es verwaltet immer nur einen Datensatz im Arbeitsspeicher. Wenn Sie im Programm auf einen Datensatz zugreifen, ruft das DataReader-Objekt diesen Datensatz von der Datenquelle ab. Es benötigt deswegen sehr wenig Ressourcen. Wenn Sie Daten nur einlesen wollen und nur einmal sequenziell durch diese Daten durchgehen wollen (was in der Praxis sehr häufig vorkommt), verwenden Sie ein DataReader-Objekt. Damit sparen Sie Ressourcen und sorgen dafür, dass Ihr Programm performanter ist, als wenn Sie ein `DataSet`-Objekt verwenden.

Wenn ein DataReader-Objekt Daten liest, kann die Verbindung so lange nicht durch andere Teile des Programms verwendet werden, bis das DataReader-Objekt geschlossen ist. Das ist für die Praxis zwar nicht unbedingt wichtig, weil die Daten normalerweise in einem »Rutsch« eingelesen werden. Behalten Sie dieses »Feature« aber im Auge.

Die wichtigsten Eigenschaften und Methoden

Die wichtigsten Eigenschaften und Methoden der DataReader-Klassen fasst Tabelle 8.5 zusammen:

Eigenschaft/Methode	Beschreibung
`bool Read()`	bewegt die aktuelle Position in der Liste auf die nächste Zeile. Diese Methode gibt `true` zurück, wenn noch weitere Zeilen folgen, ansonsten `false`.
`bool NextResult()`	Wenn der zugrunde liegende Befehl der Aufruf einer gespeicherten Prozedur (Stored Procedure) oder eine Batch-Select-Anweisung ist, verwaltet das DataReader-Objekt mehrere Datensatzlisten. Über `NextResult` können Sie zur nächsten Datensatzliste wechseln.

`object this[int i]`
 	
`object this[ string name]`
	Über diesen Indizierer erhalten Sie Zugriff auf eine Spalte der aktuellen Datenzeile. Als Index können Sie den Integer-Index der Spalte oder einen String mit dem Namen des Feldes übergeben. Die Rückgabe ist vom Typ `object`, muss also in der Regel noch konvertiert werden.

Typ `Get`*Typ*`(int i)`	Ein DataReader-Objekt besitzt eine große Anzahl Get-Methoden, über die Sie den Wert eines Feldes in Form eines echten Datentyps zurückgeben können. `GetInt32` gibt beispielsweise den Wert eines Feldes als int zurück. Sie müssen den Index des Feldes übergeben. Das ist in der Praxis oft nicht einfach ist, weil Sie oft nicht wissen, an welchem Index ein Feld in Abfrage definiert ist.
`void Close()`	Über diese Methode schließen Sie das Objekt.

Tabelle 8.5: Die wichtigsten Methoden der DataReader-Klassen

Daten lesen

Wenn Sie Daten nur einlesen und einmal durchgehen wollen, verwenden Sie idealerweise eines der DataReader-Objekte. Die Gründe dafür habe ich ja bereits genannt. Um das DataReader-Objekt erzeugen zu können, benötigen Sie zunächst ein Command-Objekt. Bei der Erzeugung dieses Objekts geben Sie im ersten Argument des Konstruktors den Befehl und im zweiten Argument das Verbindungs-Objekt an. Bei Datenbanken können Sie als Befehl eine (komplett formulierte) SQL-Abfrage, den Namen einer gespeicherten Prozedur oder den Namen einer Tabelle angeben. Wenn Sie den Namen einer gespeicherten Prozedur übergeben, müssen Sie vor der Ausführung des Befehls die Eigenschaft `CommandType` auf `CommandType.StoredProcedure` setzen. Wenn Sie eine Tabelle direkt (ohne SQL) auslesen wollen, müssen Sie hier `CommandType.TableDirect` angeben. Nicht alle Provider unterstützen diese Art Befehle. Der SQL-Server-Provider lässt beispielsweise direkte Tabellenabfragen nicht zu.

Die `ExecuteReader`-Methode des Command-Objekts gibt ein DataReader-Objekt zurück. Über dessen `Read`-Methode lesen Sie die einzelnen Datensätze ein. `Read` gibt `true` zurück, wenn noch Datensätze folgen. Über den Indizierer des DataReader-Objekts greifen Sie auf die einzelnen Felder des aktuellen Datensatzes zu.

Das folgende Beispiel liest die *Customers*-Tabelle der Northwind-Datenbank von der lokalen MSDE-Instanz ein und zeigt die Kunden-ID und den Namen des Kunden an der Konsole an:

```
/* Verbindungsobjekt erzeugen */
string conString = @"Server=(local)\NetSDK;" +
   "Trusted_Connection=Yes;Database=Northwind";
SqlConnection connection =
   new SqlConnection(conString);

/* Verbindung öffnen */
connection.Open();

/* Command-Objekt erzeugen, mit der Abfrage
 * definieren und die Verbindung übergeben */
SqlCommand command = new SqlCommand(
```

```
    "SELECT * FROM Customers", connection);

/* Abfrage ausführen */
SqlDataReader dataReader = command.ExecuteReader();

/* Daten durchgehen */
while (dataReader.Read())
{
   Console.WriteLine("{0}: {1}",
      dataReader["CustomerId"],
      dataReader["CompanyName"]);
}

/* DataReader schließen */
dataReader.Close();

/* Verbindung schließen */
connection.Close();
```

Ein DataReader-Objekt kann in Windows-Formularen nicht an Steuerelemente gebunden werden, weil es die `IList`*-Schnittstelle nicht implementiert. Das ist auch für die meisten Steuerelemente logisch, weil ein DataReader-Objekt nur ein sequenzielles Durchgehen erlaubt. Schön wäre aber, wenn man einen DataReader an Steuerelemente wie die Listbox anbinden könnte, die die Daten sowieso nur einmal einlesen. Schade. Auf ASP-NET-Webformularen können Sie ein DataReader-Objekt übrigens auch an Steuerelemente wie die Listbox binden. Scheinbar reicht den Websteuerelementen die* `IEnumerable`*-Schnittstelle aus, die von einem DataReader-Objekt implementiert wird.*

8.2.8 Daten über ein DataReader-Objekt editieren, anfügen und löschen

Wenn Sie Daten editieren, anfügen oder löschen wollen, können Sie dies mit einem `DataSet`-Objekt programmieren. Das `DataSet`-Objekt erzeugt aber einen großen Overhead, der die Ausführungsgeschwindigkeit eines Programms reduzieren kann. Für einfache Editier-, An-

füge- oder Löschvorgänge verwenden Sie besser ein DataReader-Objekt, dem Sie eine entsprechende SQL-Anweisung (`UPDATE`, `INSERT` oder `DELETE`) übergeben. Wenn Sie beispielsweise die Artikelpreise aller Artikel um 10% erhöhen wollen, macht es wenig Sinn, ein `DataSet` zu verwenden. Dazu führen Sie besser über eine `UPDATE`-Anweisung aus. Die SQL-Anweisung führen Sie dann über die Methode `ExecuteNonQuery` eines DataReader-Objekts aus. Diese Methode gibt die Anzahl der von der Anweisung betroffenen Datensätze zurück:

```
/* Verbindungsobjekt erzeugen */
string conString = @"Server=Zaphod\NetSDK;" +
   "Trusted_Connection=Yes;Database=Northwind";
SqlConnection connection =
   new SqlConnection(conString);

/* Verbindung öffnen */
connection.Open();

/* Command-Objekt erzeugen, mit der Abfrage
 * definieren und die Verbindung übergeben */
string sql = "UPDATE Products Set " +
   "UnitPrice = UnitPrice * 1.1";
SqlCommand command = new SqlCommand(sql, connection);

/* Abfrage ausführen */
int count = command.ExecuteNonQuery();

Console.WriteLine(count + " Datensätze aktualisiert");

/* Verbindung schließen */
connection.Close();
```

8.2.9 Ausführen von gespeicherten Prozeduren mit einem Command-Objekt

Mit einem Command-Objekt können Sie gespeicherte Prozeduren (Stored Procedures) ausführen und das Ergebnis der Prozedur auswerten. Dazu übergeben Sie dem Command-Objekt zunächst bei der Erzeugung den Namen der gespeicherten Prozedur. Die Prozedur

CustOrderHist in der Northwind-Datenbank gibt beispielsweise die Gesamtanzahl aller gekauften Produkte eines Kunden zurück:

```
/* Verbindung öffnen */
SqlConnection connection = new SqlConnection(
   @"Server=Zaphod\NetSDK;" +
   "Trusted_Connection=Yes;Database=Northwind");
connection.Open();

/* SqlCommand-Objekt erzeugen */
SqlCommand command = new SqlCommand("CustOrderHist",connection);
/* Definieren, dass es sich um eine
 * gespeicherte Prozedur handelt */
command.CommandType = CommandType.StoredProcedure;
```

Die meisten gespeicherten Prozeduren erwarten Parameter. Die Prozedur *CustOrderHist* erwartet im Parameter *@CustomerID* z. B. die Kunden-ID als SQL-Server-Datentyp `nchar(5)`. Für jeden Parameter müssen Sie nun ein Parameter-Objekt erzeugen. Dabei übergeben Sie den Namen des Parameters, den Datentyp (als Wert der Auflistung `SqlDbType`) und die Größe des Parameters (nur notwendig bei Stringparametern). Früher (unter ADO) war es enorm schwierig den passenden Datentyp für einen Parameter zu finden. Mit ADO.NET stellt jeder Provider eine Aufzählung der Datentypen für die Datenquelle zur Verfügung. Die Auswahl des Datentyps ist damit sehr einfach geworden.

Dem Parameter-Objekt müssen Sie dann noch mitteilen, um was für einen Parameter es sich handelt. Gespeicherte Prozeduren erlauben Parameter, die Werte in die Prozedur schreiben, über die die Prozedur Werte zurückgibt oder beides. Der Parameter der *CustOrderHist*-Prozedur ist ein einfacher Eingangsparameter:

```
/* Parameter definieren */
SqlParameter parameter = command.Parameters.Add(
    "@CustomerID", SqlDbType.NChar, 5);
parameter.Direction = ParameterDirection.Input;
```

Vergessen Sie nicht, den Wert des Parameters zu bestimmen. Das Beispiel soll die Daten des Kunden mit der Id 'ALFKI' ermitteln:

```
parameter.Value = "ALFKI";
```

CustOrderList gibt eine Datensatzliste zurück. Sie benötigen also ein DataReader-Objekt zum Lesen der Daten:

```
SqlDataReader dataReader = command.ExecuteReader();
/* und durchgehen */
while (dataReader.Read())
{
   Console.WriteLine("{0}: {1}",
      dataReader["ProductName"],
      dataReader["Total"]);
}
```

Einige gespeicherte Prozeduren geben auch einen einfachen Integer-Wert (auch parallel zur Datensatzliste) zurück. Wenn Sie diesen auswerten wollen, müssen Sie ein Parameter-Objekt mit dem Wert `ParameterDirection.ReturnValue` erzeugen. Das folgende Beispiel basiert auf der einfachen gespeicherten Demo-Prozedur *Inc*, die Sie erst noch (über ein Command-Objekt) in der Datenbank erzeugen müssen. Der SQL-Quelltext ist folgender:

```
CREATE PROCEDURE Inc(@value integer)  AS
return @value + 1
```

Die Ausführung der gespeicherten Prozedur zeigt der folgende Quelltext:

```
command = new SqlCommand("Inc", connection);
command.CommandType = CommandType.StoredProcedure;

/* Rückgabe-Parameter definieren (ohne Name) */
parameter = command.Parameters.Add("",
   SqlDbType.Int, 0);
parameter.Direction = ParameterDirection.ReturnValue;

/* Übergabe-Parameter definieren */
parameter = command.Parameters.Add(
   "@value", SqlDbType.Int, 0);
parameter.Direction = ParameterDirection.Input;
parameter.Value = 100;
```

```
/* Prozedur über ExecuteNonQuery ausführen */
command.ExecuteNonQuery();

/* und die Rückgabe auslesen */
Console.WriteLine(command.Parameters[0].Value);
```

8.2.10 Das Prinzip des DataSet

DataSet und DataTable

Ein Objekt der Klasse `DataSet`, die im Namensraum `System.Data` verwaltet wird, also für alle Provider gleich gilt, ist eine voll funktionale Datenquelle im Arbeitsspeicher. Ein `DataSet`-Objekt ist in der Lage, einen Satz von Tabellen, deren Beziehungen untereinander, Einschränkungen[3] (englisch: Constraints) und Schlüssel der Tabellen zu verwalten. Dieses Objekt ist nicht mit einer physikalischen Datenquelle verbunden, sondern verwaltet alle Daten im Arbeitsspeicher. Das unterscheidet ein solches Objekt von anderen Techniken wie ADO, die (normalerweise) mit einer externen Datenquelle verbunden sind und diese lediglich repräsentieren.

Sie können mit einem `DataSet`-Objekt Datenbanken im Arbeitsspeicher dynamisch erzeugen (und dann beispielsweise in eine XML-Datei speichern) oder die Daten aus einer externen Datenquelle abfragen und daraus die Arbeitsspeicher-Datenbank bilden. Das `DataSet`-Objekt erlaubt das Verändern der Daten und das Zurückschreiben der geänderten Daten in die Datenquelle.

Tabelle 8.6 zeigt die wichtigsten Eigenschaften und Methoden der `DataSet`-Klasse.

3. Eine Einschränkung ist eine Regel, die das Editieren von Daten nur unter bestimmten Einschränkungen zulässt. Eine (Prüf-)Einschränkung auf einem Feld *Artikelpreis* kann beispielsweise dafür sorgen, dass kein negativer Preis eingegeben werden kann.

Eigenschaft/ Methode	Bedeutung
`Tables`	Über diese Eigenschaft erhalten Sie Zugriff auf die verwalteten Tabellen. `Tables` ist eine Auflistung von `DataTable`-Objekten. Ein `DataTable`-Objekt repräsentiert eine einzelne Tabelle. Über Methoden und Eigenschaften dieses Objekts können Sie die Daten bearbeiten.
`Relations`	Diese Eigenschaft verwaltet die Beziehungen zwischen den Tabellen in Form von einzelnen `DataRelation`-Objekten.
`void WriteXml(...)`	`WriteXml` erzeugt aus den Daten eine XML-Datei mit Schema-Definition. Im ersten Argument können Sie einen Dateinamen, ein Stream-, ein TextWriter- oder ein XmlWriter-Objekt übergeben.
`XmlReadMode ReadXml(...)`	Diese Methode liest eine XML-Datei mit Schema-Definition ein. Im ersten Argument können Sie einen Dateinamen, ein Stream-, ein TextReader- oder ein XmlReader-Objekt übergeben.
`string GetXml()`	`GetXml` gibt die gespeicherten Daten als XML-String zurück.

Tabelle 8.6: Die wichtigsten Eigenschaften und Methode der DataSet-Klasse

Eine Update-Methode zum Aktualisieren der Daten in der Datenquelle suchen Sie hier übrigens vergeblich (falls Sie diese suchen ...). Diese Methode gehört zu einem DataAdapter-Objekt.

Ein `DataTable`-Objekt repräsentiert eine Tabelle innerhalb des `DataSet`. Über die Tables-Auflistung erhalten Sie Zugriff auf diese einzelnen `DataTable`-Objekte. Dem Indizierer dieser Auflistung können Sie dazu entweder den Index oder den Namen der Tabelle übergeben. Ein `DataTable`-Objekt besitzt die in Tabelle 8.7 dargestellten wichtigen Eigenschaften und Methoden. Einige dieser Methoden werden Sie allerdings wohl erst verstehen, wenn Sie den Abschnitt »Daten im Programm mit dem DataSet editieren« ab Seite 415 gelesen haben.

Eigenschaft/Methode	Bedeutung
`void AcceptChanges()`	schreibt Änderungen, die an einzelnen Datensätzen vorgenommen wurden, in das `DataSet`. Diese Methode wird implizit von der `Update`-Methode eines DataAdapter-Objekts aufgerufen (siehe auf Seite 408).
`void RejectChanges()`	verwirft alle Änderungen, die an einzelnen Datensätzen vorgenommen wurden
`Columns`	`Columns` ist eine Auflistung der Spalten, die diese Tabelle besitzt, in Form von `DataColumn`-Objekten.
`Constraints`	Ein `DataTable`-Objekt kann verschiedene Einschränkungen (englisch: Constraints) verwalten. Einschränkungen definieren Regeln für das Editieren, die nicht verletzt werden können. Über die `Constraints`-Auflistung erreichen Sie die Einschränkungen. Über die `Add`-Methode fügen Sie neue Einschränkungen hinzu.
`ChildRelations` `ParentRelations`	Die einzelnen Tabellen in einem DataSet können miteinander in Beziehung gesetzt werden. In einer Buchverwaltungsdatenbank verweist das Feld *AutorId* der Buchtabelle beispielsweise auf das Feld *Id* der Autorentabelle. Über die Eigenschaften `ChildRelations` und `ParentRelations` können Sie die Beziehungen einer Tabelle zu anderen Tabellen ermitteln oder auch definieren.
`PrimaryKey`	`PrimaryKey` ist ein Array aus `DataColumn`-Objekten (Spalten), die den Primärschlüssel der Tabelle spezifizieren.
`Rows`	Über diese Auflistung erreichen Sie die gespeicherten Datensätze in Form von `DataRow`-Objekten.
`TableName`	`TableName` verwaltet den Namen der Tabelle.

Tabelle 8.7: Die wichtigsten Eigenschaften und Methoden der DataTable-Klasse

Über die Rows-Eigenschaft können Sie einzelne Datensätze einer Tabelle bearbeiten, beispielsweise Daten auslesen, hinzufügen oder löschen. Ein DataRow-Objekt besitzt dazu die in Tabelle 8.8 dargestellten wichtigen Eigenschaften und Methoden.

Eigenschaft/ Methode	Bedeutung
RowState	spezifiziert den aktuellen Status der Zeile. RowState ist eine Aufzählung vom Typ DataRowState mit den Werten Added (Zeile ist neu), Deleted (Zeile wurde gelöscht), Detached (Zeile ist noch kein Teil der Tabelle, sie wurde gerade erst neu erzeugt), Modified (die Daten der Zeile wurden geändert, aber noch nicht in die aktuellen Werte der Zeile geschrieben) und Unchanged (Zeile ist seit dem letzten Speichern unverändert).
object Item(...) object this[...]	über die Item-Eigenschaft, die gleichzeitig der Indizierer der Klasse ist, können Sie den Wert einer einzelnen Spalte auslesen.
void BeginEdit()	Über diese Methode können Sie einen Editiervorgang einleiten, den Sie dann nach dem Editieren mit EndEdit oder CancelEdit beenden.
void CancelEdit()	bricht das Editieren ab und verwirft die neuen Daten.
void EndEdit()	beendet das Editieren. Wird implizit von AcceptChanges aufgerufen.
void Delete()	löscht eine Zeile.
void AcceptChanges()	Nach dem Editieren einer Zeile können Sie die neuen Daten über diese Methode in das DataSet schreiben. Alternativ können Sie die AcceptChanges-Methode des DataTable-Objekts aufrufen, um die Änderungen an mehreren Zeilen in das DataSet zu schreiben.
void RejectChanges()	verwirft alle Änderungen an der Zeile.

Tabelle 8.8: Die wichtigsten Eigenschaften und Methoden der DataRow-Klasse

Ein DataColumn-Objekt, das von der Item-Eigenschaft bzw. vom Indizierer des DataRow-Objekts zurückgegeben wird, besitzt die in Tabelle 8.9 angegebenen wichtigen Eigenschaften, über die Sie das Verhalten einer Spalte spezifizieren können.

Eigenschaft	Bedeutung
AllowDBNull	spezifiziert, ob der Wert DBNull (der für ein leeres Feld steht) in die Spalte geschrieben werden kann.
AutoIncrement	bestimmt, ob der (nummerische) Wert der Spalte bei neuen Datensätzen automatisch um Eins hochgezählt wird.
AutoIncrement-Seed	bestimmt den Startwert für das automatische Hochzählen.
Caption	bestimmt eine Beschriftung der Spalte, die unabhängig vom Namen sein kann.
ColumnName	speichert den Namen der Spalte.
DataType	bestimmt den Datentyp mit einem der in System.Type definierten Typen.
ReadOnly	bestimmt, ob die Spalte schreibgeschützt ist.
Unique	bestimmt, ob der Wert der Spalte bezogen auf alle Zeilen eindeutig sein muss.

Tabelle 8.9: Die wichtigsten Eigenschaften der DataColumn-Klasse

DataSet dynamisch erzeugen

Ich denke, Sie verstehen das Grundprinzip eines DataSet-Objekts am besten, wenn Sie einfach eines dynamisch erzeugen. Die so erzeugte »Datenbank« soll Buchdaten verwalten, also Autoren- und Buchtitel. Abbildung 8.3 zeigt das Schema dieser Datenbank.

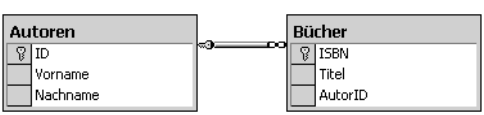

Bild 8.3: Die Beispiel-Datenbank

Der folgende Quelltext erzeugt ein DataSet mit den benötigten Tabellen, die miteinander in Beziehung gesetzt und mit Daten gefüllt werden:

```
/* Variable für das DataSet */
DataSet bookDataSet = new DataSet("Buchverwaltung");

/* Die Struktur des DataSet definieren */

/* Autoren-Tabelle erzeugen */
DataTable authorTable = bookDataSet.Tables.Add("Autoren");

/* Spalten anfügen */
authorTable.Columns.Add("ID",typeof(int));
authorTable.Columns.Add("Vorname",typeof(string));
authorTable.Columns.Add("Nachname",typeof(string));

/* Primärschlüssel definieren */
authorTable.PrimaryKey =
   new DataColumn[] {authorTable.Columns["ID"]};

/* Bücher-Tabelle erzeugen */
DataTable bookTable = bookDataSet.Tables.Add("Bücher");

/* Spalten anfügen */
bookTable.Columns.Add("ISBN",typeof(string));
bookTable.Columns.Add("Title",typeof(string));
bookTable.Columns.Add("AutorId",typeof(int));

/* Primärschlüssel definieren */
bookTable.PrimaryKey = new DataColumn[]
   {bookTable.Columns["ISBN"]};

/* Beziehungen zwischen den Tabellen festlegen */
bookDataSet.Relations.Add("Autoren-Bücher",
   authorTable.Columns["ID"],
   bookTable.Columns["AutorId"]);
```

Das so erzeugte `DataSet`-Objekt können Sie an Steuerelemente anbinden um dem Benutzer das Editieren, Anfügen und Löschen zu ermöglichen.

In den Buchbeispielen finden Sie ein Projekt, das die Buchverwaltung implementiert, das Editieren ermöglicht, die Daten in eine XML-Datei abspeichert und auch wieder aus der XML-Datei einliest.

Sie können Datensätze auch im Programm hinzufügen, ändern und löschen. Wie Sie das programmieren, zeige ich ab Seite 413.

Das `DataSet`-Objekt überprüft alle definierten Einschränkungen und Regeln beim Editieren. Wenn Sie beispielsweise versuchen, einen Autor mit einer Autoren-ID anzufügen, die bereits existiert, erzeugt das DataSet eine Ausnahme (weil die Autoren-ID der Primärschlüssel ist). Sie können auch kein Buch anfügen oder ändern und dabei eine Autoren-ID übergeben, die nicht existiert oder einen Autor löschen, der noch in der Buch-Tabelle referenziert wird. Wenn Sie Datenbindung verwenden, werden diese Regeln natürlich auch eingehalten. Sie müssen lediglich eine Ausnahmebehandlung vorsehen. Das ist ein enormer Fortschritt gegenüber älteren Datenbearbeitungstechnologien.

Daten in XML-Dateien verwalten

Das `DataSet`-Objekt verwaltet seine Daten intern in XML-Form. Deswegen ist es möglich, diese einfach in eine XML-Datei zu speichern:

```
bookDataSet.WriteXml(@"C:\Buchverwaltung.xml",
   XmlWriteMode.WriteSchema);
```

Wenn Sie im zweiten Argument `XmlWriteMode.WriteSchema` angeben, wird die gesamte Struktur der Datenbank (das so genannte Schema) in Form einer XSD-Schemadefinition gleich mit in die XML-Datei geschrieben. Das besitzt den enormen Vorteil, dass die definierten Einschränkungen (Primärschlüssel, Beziehungen etc.) nicht verloren gehen. Sie können die Schemadefinition auch weglassen:

```
bookDataSet.WriteXml(@"C:\Buchverwaltung.xml",
   XmlWriteMode.IgnoreSchema);
```

Wenn Sie diese XML-Datei dann wieder einlesen, müssen Sie die Struktur nicht mehr im `DataSet`-Objekt definieren:

```
bookDataSet = new DataSet();
bookDataSet.ReadXml(@"C:\Pubs.xml");
```

Enthält die XML-Datei das Schema der »Datenbank«, werden alle Einschränkungen und Regeln automatisch übernommen. So können Sie dem Anwender eine Datenbank zur Verfügung stellen, ohne eine ADO.NET-Datenquelle dafür zu verwenden.

Das Beispiel im SQL Server

Um zu erläutern, wie Sie mit dem DataSet eine »echte« Datenquelle bearbeiten, verwende ich eine möglichst einfache Datenbank. Diese Datenbank soll wie das Beispiel auf Seite 404 Autorendaten und Buchtitel speichern. Diese Datenbank müssen Sie zunächst erzeugen. Ich zeige hier einfach, wie Sie das über SQL erledigen können. Nebenbei lernen Sie gleich noch, wie Sie Datenbanken im Programm dynamisch erzeugen können:

```
/* Verbindung zum SQL Server ohne Datenbank
 * aufbauen */
SqlConnection connection = new SqlConnection(
   @"Server=Zaphod\NetSDK;" +
   "Trusted_Connection=Yes;Database=");
connection.Open();

/* Neue Datenbank erzeugen */
string sql = "CREATE DATABASE Buchverwaltung";
SqlCommand command = new SqlCommand(sql, connection);
command.ExecuteNonQuery();

/* Verbindung wieder schließen */
connection.Close();

/* Mit der neuen Datenbank verbinden */
connection = new SqlConnection(
   @"Server=Zaphod\NetSDK;" +
   "Trusted_Connection=Yes;Database=Buchverwaltung");
connection.Open();
```

```
/* Tabellen erzeugen */
sql = "CREATE Table Autoren (ID integer PRIMARY KEY " +
   ", Vorname nvarchar(50) , Nachname nvarchar(50))";
command = new SqlCommand(sql, connection);
command.ExecuteNonQuery();

sql = "CREATE Table Bücher (ISBN nvarchar(50) " +
  " PRIMARY KEY, Titel nvarchar(50), AutorID integer)";
command = new SqlCommand(sql, connection);
command.ExecuteNonQuery();

/* Beziehung definieren */
sql = "ALTER Table Bücher ADD CONSTRAINT FK_Bücher " +
   "FOREIGN KEY (AutorID) REFERENCES Autoren(ID)";
command = new SqlCommand(sql, connection);
command.ExecuteNonQuery();

/* Verbindung schließen */
connection.Close();
```

Beachten Sie, dass sinnvolle Fehlermeldungen nur dann ausgegeben werden, wenn Sie eine Ausnahmebehandlung vorsehen. Ohne Ausnahmebehandlung meldet Visual Studio lediglich immer nur einen »Systemfehler«.

Wenn Sie diesen Quellcode ausprobieren, und aufgrund einer fehlerhaften Ausführung die Datenbank wieder löschen müssen, können Sie dazu die SQL-Anweisung »DROP DATABASE Buchverwaltung« verwenden.

8.2.11 DataAdapter

Wenn Sie ein `DataSet`-Objekt aus einer Datenquelle erzeugen wollen, benötigen Sie pro Tabelle, die im `DataSet`-Objekt verwaltet wird, ein DataAdapter-Objekt. DataAdapter-Objekte sind Bindeglieder zwischen dem Connection- und dem `DataSet`-Objekt. Ein DataAdapter führt die Abfrage der Struktur und der Daten einer Datenquelle und das Zurückschreiben geänderter Daten aus. Über die `Update`-Methode schreiben Sie die in DataTable-Objekten geänderten Daten in

die Datenquelle zurück. Ein DataAdapter beinhaltet vier DataCommand-Objekte, je eines zum Abfragen, Anfügen, Editieren und Löschen von Daten, die Sie über folgende Eigenschaften erreichen:

- `SelectCommand`: wird für die Abfrage der Daten verwendet.
- `DeleteCommand`: wird verwendet, wenn Datensätze gelöscht werden müssen.
- `UpdateCommand`: wird verwendet, wenn Datensätze geändert werden müssen.
- `InsertCommand`: wird verwendet, wenn Datensätze angefügt werden sollen.

Wenn Sie ein DataAdapter-Objekt erzeugen, geben Sie im Konstruktor den Befehl für die Abfrage der Daten und die Verbindung an. Die folgende Anweisung erzeugt ein `SqlDataAdapter`-Objekt für den Zugriff auf die Autoren-Tabelle unserer Beispieldatenbank (der Vollständigkeit halber inklusive Öffnen der Verbindung):

```
SqlConnection connection = new SqlConnection(
   @"Server=Zaphod\NetSDK;" +
   "Trusted_Connection=Yes;Database=Buchverwaltung");
connection.Open();
SqlDataAdapter dataAdapter = new SqlDataAdapter(
   "SELECT * FROM Autoren", connection);
```

Damit haben Sie das `SelectCommand`-Objekt definiert, aber noch nicht die anderen Command-Objekte. Wenn Sie Daten aktualisieren, einfügen oder löschen wollen, müssen Sie diese anderen Objekte ebenfalls oder ein CommandBuilder-Objekt einsetzen. Wenn Sie ein CommandBuilder-Objekt erzeugen und diesem im Konstruktor das DataAdapter-Objekt übergeben, trägt sich das CommandBuilder-Objekt als Ereignisbehandler des `RowUpdating`-Ereignis des DataAdapter-Objekts ein und übergibt dem `Command`-Argument des Ereignisarguments den dynamisch erzeugten Befehl. Puh. Warum so kompliziert, Microsoft?

Na ja, es funktioniert:

```
SqlCommandBuilder commandBuilder =
  new SqlCommandBuilder(dataAdapter);
```

Die SQL-Befehle, die das CommandBuilder-Objekt erzeugt, sind aber alles andere als performant. Sie können diese Befehle über die Methoden `GetUpdateCommand`, `GetInsertCommand` und `GetDeleteCommand` auslesen. Der für das Aktualisieren eines Autors erzeugte Befehl sieht z. B. so aus:

```
UPDATE Autoren SET ID = @p1 , Vorname = @p2 ,
 Nachname = @p3 WHERE ( (ID = @p4) AND
 ((Vorname IS NULL AND @p5 IS NULL) OR (Vorname = @p6))
  AND ((Nachname IS NULL AND @p7 IS NULL)
 OR (Nachname = @p8)))
```

Die mit @ bezeichneten Parameter werden bei der Ausführung des Befehls durch die Daten im Datensatz ersetzt.

Wenn Sie die Befehle für die Aktualisierungen selbst definieren, besitzen Sie eine enorme Flexibilität (aber leider auch viel Arbeit). Sie können hochoptimierte Befehle erzeugen oder sogar für das Aktualisieren, Anfügen und Löschen gespeicherte Prozeduren verwenden. Als Beispiel will ich hier nur den Befehl zum Anfügen eines Autors definieren:

```
dataAdapter.InsertCommand = new SqlCommand(
   "INSERT INTO Autoren (ID, Vorname, Nachname) " +
   "VALUES (@Id, @Vorname, @Nachname)", connection);
```

Vergessen Sie nicht, dem Befehl die Verbindung zu übergeben. Ohne Verbindung erhalten Sie beim Update keine Ausnahme, aber leider auch keine Änderung in der Datenbank.

Die Parameter müssen Sie dann noch spezifizieren:

```
dataAdapter.InsertCommand.Parameters.Add(
   "@Id", SqlDbType.Int, 0, "ID");
dataAdapter.InsertCommand.Parameters.Add(
   "@Vorname", SqlDbType.NVarChar, 50, "Vorname");
dataAdapter.InsertCommand.Parameters.Add(
   "@Nachname", SqlDbType.NVarChar, 50, "Nachname");
```

Im ersten Argument übergeben Sie den Namen des Parameters, im zweiten den Datentyp des Feldes, im dritten die Größe des Feldes (falls notwendig) und im vierten den Namen des Datenfeldes.

Füllen des DataSet

Wenn Sie das `DataAdapter`-Objekt erzeugt und definiert haben, können Sie damit ein `DataSet`-Objekt füllen:

```
/* DataSet erzeugen */
DataSet dataSet = new DataSet();

/* DataSet mit dem Ergebnis der Abfrage füllen */
dataAdapter.Fill(dataSet, "Autoren");
```

Im zweiten Argument müssen Sie den Namen der Quelltabelle übergeben, damit Sie diese über die `Tables`-Auflistung des `DataSet` identifizieren können.

Nach dem Füllen sollten bzw. müssen Sie die speziellen Eigenschaften der Felder definieren. Das Feld *ID* ist beispielsweise der Primärschlüssel:

```
/* Primärschlüsselspalte definieren */
dataSet.Tables["Autoren"].PrimaryKey = new DataColumn[]
   {dataSet.Tables["Autoren"].Columns["ID"]};
```

Ich habe bisher kein Weg gefunden, die speziellen Eigenschaften (wie z. B. automatisches Inkrementieren des Werts) und Einschränkungen (englisch: Constraints) der Datenfelder (wie Primärschlüssel- oder Prüfeinschränkungen) automatisch aus der Datenbank zu ermitteln. Ich finde es ziemlich eigenartig, dass ein DataAdapter-Objekt diese nicht automatisch ausliest. Berücksichtigen muss ich die speziellen Eigenschaften und Einschränkungen doch auf jeden Fall. Beim Update der Daten wird die Datenbank Datensätze, die irgendwelche Einschränkungsregeln verletzen, nicht akzeptieren und eine Ausnahme erzeugen.

8.2.12 Datenbindung mit dem DataSet

Ein `DataTable`-Objekt können Sie an beliebige Steuerelemente binden, die Datenbindung erlauben. Damit können Sie dem Benutzer beispielsweise das einfache Editieren ermöglichen. Die Grundlagen

entsprechen denen, die bereits ab Seite 371 behandelt wurden. Sie binden natürlich nun ein `DataTable`-Objekt an die Steuerelemente. Der folgende Quelltext bindet die Autoren-Tabelle an drei Textboxen:

```
txtID.DataBindings.Add("Text",
   dataSet.Tables["Autoren"], "Id");
txtFirstName.DataBindings.Add("Text",
   dataSet.Tables["Autoren"], "Vorname");
txtLastName.DataBindings.Add("Text",
   dataSet.Tables["Autoren"], "Nachname");
```

Wenn Sie Datenbindung einsetzen, muss das `DataSet`*- und das DataAdapter-Objekt auf Klassenebene (*`private`*) deklariert sein, damit Sie im Formular damit arbeiten können. Die Verbindungsvariable sollte ebenfalls privat deklariert sein, damit Sie diese im Formular wiederverwenden können. Ob Sie die Verbindung nach dem Erzeugen des* `DataReader`*-Objekts schließen und dann für jedes Update wieder neu öffnen und danach wieder schließen, oder ob Sie die Verbindung erst im* `Closing`*-Ereignis schließen, bleibt ganz Ihnen überlassen. Ressourcenschonender ist die erste Variante. Die zweite kann u. U. für den Anwender nervig werden, wenn Sie diesem einen Speichern-Button zur Verfügung stellen. Der Verbindungsaufbau kann nach dem Ablauf der Connection-Pooling-Zeit (normalerweise 60 Sekunden) schon recht lange dauern.*

Über den bereits bekannten Bindungskontext des Formulars können Sie dem Anwender das Positionieren, das Editieren, das Erzeugen neuer Datensätze und das Löschen erlauben. Sie müssen nun aber dafür sorgen, dass die geänderten Daten in die Datenquelle zurückgeschrieben werden. Das erledigen Sie dann zumindest im `Closing`-Ereignis des Formulars. Alternativ können Sie dem Anwender natürlich auch einen Button zum Aktualisieren der Datenquelle zur Verfügung stellen. Die Aktualisierung führen Sie über die `Update`-Methode des `DataAdapter`-Objekts aus, der Sie das `DataSet`-Objekt und den Namen der Tabelle übergeben. Dabei müssen Sie beachten, dass der Anwender den aktuellen Datensatz vielleicht gerade noch bearbeitet

hat. Rufen Sie einfach vor dem Update die `EndCurrentEdit`-Methode des Bindungskontext-Objekts auf, um diesen Editiervorgang zu beenden (und damit die Änderungen zu übernehmen):

```
private void StartForm_Closing(object sender,
   System.ComponentModel.CancelEventArgs e)
{
   /* einen eventuellen aktuellen Editiervorgang
    * beenden, damit die Änderungen am aktuellen
    * Datensatz auch zurückgeschrieben werden */
    this.BindingContext[
       dataSet.Tables["Autoren"]].EndCurrentEdit();

   /* Änderungen am DataSet in die Datenbank
    * zurückschreiben */
   dataAdapter.Update(dataSet, "Autoren");

   /* Verbindung zur Datenbank schließen */
   connection.Close();
}
```

Den kompletten Quellcode zur Datenbindung mit dem `DataSet`*-Objekt finden Sie (natürlich) in den Buchbeispielen.*

8.2.13 Mehrere Tabellen im DataSet

Wenn Sie mehrere Tabellen im `DataSet` verwalten wollen, benötigen Sie normalerweise für jede Tabelle ein DataAdapter-Objekt. Der Grund dafür ist, dass die Insert-, Update- und Delete-Befehle für die einzelnen Tabellen unterschiedlich sind. Leider müssen Sie die Einschränkungen und die Beziehungen der Tabellen nach dem Einlesen der Daten selbst definieren. Das `DataSet` macht dies nicht automatisch. Das folgende Beispiel liest die Autoren- und die Büchertabelle ein, definiert die Primärschlüssel und die Beziehung zwischen den Tabellen:

```
/* Verbindung zur SQL-Server-Datenbank
 * 'Buchverwaltung' herstellen */
SqlConnection connection = new SqlConnection(
```

```
   @"Server=Zaphod\NetSDK;" +
   "Trusted_Connection=Yes;Database=Buchverwaltung");
connection.Open();

/* DataAdapter erzeugen */
SqlDataAdapter authorAdapter = new
   SqlDataAdapter("SELECT * FROM Autoren", connection);
SqlDataAdapter bookAdapter = new
   SqlDataAdapter("SELECT * FROM Bücher", connection);

/* CommandBuilder für die Befehle erzeugen */
new SqlCommandBuilder(authorAdapter);
new SqlCommandBuilder(bookAdapter);

/* DataSet erzeugen */
DataSet dataSet = new DataSet();

/* DataSet mit dem Ergebnis der Abfragen füllen */
authorAdapter.Fill(dataSet, "Autoren");
bookAdapter.Fill(dataSet, "Bücher");

/* Referenz auf die Tabellen holen */
DataTable authorTable = dataSet.Tables["Autoren"];
DataTable bookTable = dataSet.Tables["Bücher"];

/* Primärschlüsselspalten definieren */
authorTable.PrimaryKey = new DataColumn[]
    {dataSet.Tables["Autoren"].Columns["ID"]};
bookTable.PrimaryKey = new DataColumn[]
    {dataSet.Tables["Bücher"].Columns["ISBN"]};

/* Beziehungen definieren */
dataSet.Relations.Add("Autoren-Bücher",
   authorTable.Columns["ID"],
   bookTable.Columns["AutorID"]);
```

Das sieht zwar nach Arbeit aus. Sie besitzen damit aber alle Möglichkeiten. So können Sie für Datenbanken, die keine Beziehungen erlauben, solche definieren und sogar Tabellen aus unterschiedlichen

Datenquellen miteinander in Beziehung setzen (was vor ADO.NET nicht möglich war). Wollen Sie in Ihrem Programm dafür sorgen, dass ein Feld nur eindeutige Werte zulässt, obwohl das in der Datenquelle gar nicht der Fall ist? Kein Problem, setzen Sie einfach die Eigenschaft `Unique` der entsprechenden Spalte auf `true`. Diese Flexibilität ist schon fast unheimlich ...

8.2.14 Daten im Programm mit dem DataSet durchgehen

Wenn Sie die Daten eines `DataSet` durchgehen wollen, können Sie dies einfach über einen `Integer`-Index realisieren. Über den Indizierer des `DataRow`-Objekts, das Sie über den Indizierer des `DataTable`-Objekts erreichen, können Sie die einzelnen Spalten auslesen:

```
/* Autoren-Tabelle sequenziell durchgehen */
DataTable authorTable = dataSet.Tables["Autoren"];
for (int i=0; i<authorTable.Rows.Count; i++)
{
   MessageBox.Show(
      authorTable.Rows[i]["Vorname"] + " " +
      authorTable.Rows[i]["Nachname"]);
}
```

8.2.15 Daten im Programm mit dem DataSet editieren

Das Prinzip

Wenn Sie Daten im Programm mit einem `DataSet`-Objekt editieren wollen, greifen Sie über die `Rows`-Eigenschaft eines `DataTable`-Objekts auf die Daten zu. Das Anfügen, Ändern und Löschen erfolgt nach einem komplexen Prinzip. Ein `DataRow`-Objekt verwaltet dazu zunächst seinen aktuellen Status in der Eigenschaft `RowState` mit den folgenden Werten:

- `Added`: Die Zeile wurde neu hinzugefügt und `AcceptChanges` wurde bisher nicht aufgerufen.
- `Deleted`: Die Zeile wurde gelöscht.
- `Detached`: Die Zeile gehört nicht zu einem `DataTable`-Objekt.
- `Modified`: Die Zeile wurde geändert und `AcceptChanges` wurde bisher nicht aufgerufen.

- `Unchanged`: Die Zeile wurde seit dem Abruf aus der Datenquelle oder seit dem letzten Aufruf von `AcceptChanges` nicht geändert.

Neben dem Status verwaltet eine Zeile vier Wertesätze:

- die originalen Werte, die aktuell im `DataSet` gespeichert sind (und folglich angezeigt werden, wenn die Daten irgendwo ausgegeben werden),
- die in einem Editiervorgang gerade aktuellen Daten,
- Daten, die in einem mit `BeginEdit` eingeleiteten Editiervorgang noch nicht als aktuelle Daten abgelegt wurden (so genannte vorgeschlagene Daten) und
- Defaultdaten (die im Schema des DataSet als solche eingetragen sind).

Beim Editieren von Daten müssen Sie unterscheiden, ob das `DataSet` über einen DataAdapter eine Datenquelle bearbeitet oder ob das `DataSet` wie im Beispiel auf Seite 404 ohne Datenquelle verwenden.

Wenn Sie in einem `DataSet` mit Datenquelle Daten angefügt, geändert oder gelöscht haben, müssen Sie diese Änderungen in die Datenquelle zurückschreiben. Dazu verwenden Sie die `Update`-Methode des DataAdapter-Objekts:

```
authorAdapter.Update(dataSet, "Autoren");
```

Die `Update`-Methode geht alle Datensätze durch, überprüft den Status der Datensätze und verwendet das entsprechende Command-Objekt, um die Datensätze zu aktualisieren, anzufügen oder zu löschen. Für die SQL-Anweisungen werden dann je nach Bedarf die aktuellen Werte (für Aktualisierungen) und die originalen Werte (für die Identifizierung geänderter oder gelöschter Datensätze) verwendet.

Wenn Sie in einem `DataSet` ohne Datenquelle Daten angefügt, geändert oder gelöscht haben, müssen Sie stattdessen `AcceptChanges` aufrufen. Diese Methode können Sie bei einem `DataRow`-Objekt, einem `DataTable`-Objekt oder einem `DataSet`-Objekts aufrufen, um nur eine Zeile oder gleich mehrere Zeilen gemeinsam zu aktualisieren. Mit `RejectChanges` können Sie die Änderungen verwerfen.

Die `Update`-Methode eines DataAdapter-Objekts ruft nach dem erfolgreichen Aktualisieren der Datenquelle implizit `AcceptChanges` auf, um den Status der Datensätze im `DataSet`-Objekt auf `Unchanged` zu setzen.

Sie dürfen `AcceptChanges` *nicht aufrufen, wenn das* `DataSet` *Daten aus einer Datenquelle verwaltet.* `AcceptChanges` *setzt den Status der Zeilen auf* `Unchanged` *zurück. Die* `Update`*-Methode des DataAdapter-Objekts erkennt aber am Status, ob die Zeilen aktualisiert, angefügt oder gelöscht werden müssen. Ist der Status* `Unchanged`*, wird die Zeile nicht geändert, angefügt bzw. gelöscht. Ich habe das »ausprobiert« (in Wirklichkeit habe ich etwa zwei Stunden nach dem Fehler gesucht ...). Das ist leider nur sehr spärlich dokumentiert ...*

Die folgenden Ausführungen sollen das Prinzip beim Editieren verdeutlichen:

Wenn Sie einen Datensatz anfügen, beispielsweise einen Autoren:

```
dataSet.Tables["Autoren"].Rows.Add(
  new object[] {1, "J.","Irving"});
```

schalten Sie den Status der Zeile um in `Added` und beschreiben die aktuellen Daten. Wird danach `AcceptChanges` aufgerufen (implizit bei einem `DataSet` mit Datenquelle, explizit bei einem `DataSet` ohne Datenquelle), werden die aktuellen Daten in die originalen Daten übertragen und der Status der Zeile auf `Unchanged` gesetzt. Wurde `AcceptChanges` über die `Update`-Methode eines `DataAdapter`-Objekts implizit aufgerufen, wurden die angefügten Daten natürlich vorher auch in die Datenquelle geschrieben.

Wenn Sie eine Zeile aktualisieren, beispielsweise die erste:

```
dataSet.Tables["Autoren"].Rows[0]["Id"] = 1001;
dataSet.Tables["Autoren"].Rows[0]["Vorname"] = "John";
```

schalten Sie den Status der Zeile auf `Modified` um. Die Zeile verwaltet dann aktuelle Werte (Id=1001 und Vorname="John") und die originalen Werte, die vor dem Editieren gültig waren (Id=1 und Vorname="J.").

Wird dann AcceptChanges aufgerufen, werden die aktuellen Werte in die originalen kopiert und der Status auf Unchanged gesetzt.

Sie können beim Editieren mit einem DataSet ohne Datenquelle noch einen anderen Weg gehen, indem Sie vor dem Editieren die BeginEdit-Methode des DataRow-Objekts aufrufen. Wenn Sie dann Daten in die Zeile schreiben, werden diese nicht in die aktuellen Daten übernommen, sondern zunächst in die vorgeschlagenen Daten. Erst wenn Sie EndEdit aufrufen, werden die vorgeschlagenen Daten in die aktuellen kopiert. Mit CancelEdit können Sie die vorgeschlagenen Daten verwerfen. Nach EndEdit muss wieder AcceptChanges aufgerufen werden, damit die aktuellen Daten in die originalen Daten kopiert werden.

Sie können die spezifische Version der Daten über den Indizierer des DataRow-Objekts abfragen, indem Sie im zweiten Argument einen Wert der Aufzählung DataRowVersion übergeben. Die möglichen Werte dieser Aufzählung stehen für die Version der Daten: Original (der originale Wert im DataSet), Current (der aktuelle Wert beim Editieren), Default (der Defaultwert der Datenquelle) oder Proposed (der vorgeschlagene Wert).

Kompliziert? Finde ich auch. Das Ganze soll Ihnen wohl möglichst viel Flexibilität beim Editieren geben. So können Sie bei einer Fehleingabe durch einfaches Abbrechen die originalen Daten »wiederherstellen« (was dann ja eigentlich gar nicht der Fall ist) und auf verschiedene Versionen der Daten zurückgreifen (vielleicht um dem Benutzer bei einer Fehleingabe die originalen Daten einer einzelnen Spalte vorzuschlagen). Beachten Sie aber, dass Sie nicht immer auf die verschiedenen Versionen der Daten zurückgreifen können. Bei einer neuen, unmodifizierten Zeile steht beispielsweise nur der originale Satz zur Verfügung. Wird die Zeile editiert, erhalten Sie zudem Zugriff auf den aktuellen Satz.

Datensätze hinzufügen

Über die Add-Methode der Rows-Auflistung können Sie Zeilen hinzufügen. Sie können dieser Methode im Konstruktor ein object-Array mit den Werten übergeben:

```
/* Autoren anfügen */
dataSet.Tables["Autoren"].Rows.Add(
```

```
  new object[] {1, "John","Irving"});
dataSet.Tables["Autoren"].Rows.Add(
  new object[] {2, "Matt", "Ruff"});
```

Alternativ können Sie eine Zeile separat erzeugen, die Spalten einzeln beschreiben und dann übergeben. Der folgende Quellode fügt einige Bücher an:

```
DataTable bookTable = dataSet.Tables["Bücher"];

DataRow newRow = bookTable.NewRow();
newRow["ISBN"] = "3257224451";
newRow["Titel"] =
  "Die wilde Geschichte vom Wassertrinker";
newRow["AutorId"] = 1;
bookTable.Rows.Add(newRow);

newRow = bookTable.NewRow();
newRow["ISBN"] = "3423117370";
newRow["Titel"] = "Fool on the Hill";
newRow["AutorId"] = 2;
bookTable.Rows.Add(newRow);

newRow = bookTable.NewRow();
newRow["ISBN"] = "342312721X";
newRow["Titel"] = "G.A.S.";
newRow["AutorId"] = 2;
bookTable.Rows.Add(newRow);
```

Wenn Sie beim Hinzufügen eine Regel bzw. Einschränkung verletzen oder einen ungültigen Datentyp übergeben, generiert das `DataSet`-Objekt eine Ausnahme.

Datensätze ändern

Das Ändern von Datensätzen ist recht einfach. Sie beschreiben einfach die Spalten des Datensatzes, der geändert werden soll. Das folgende Beispiel ändert den (unvollständigen) Titel des dritten Buchs:

```
DataRow row = bookTable.Rows[2];
row["Titel"] = "G.A.S. Die Trilogie der Stadtwerke.";
```

Beim Ändern berücksichtigt das `DataSet` natürlich alle Regeln und erzeugt eine Ausnahme, wenn eine Regel verletzt wird.

Vor dem Ändern müssen Sie den oder die Datensätze normalerweise erst suchen. Wie das geht, zeige ich ab Seite 422.

Datensätze löschen

Das Löschen von Datensätzen ist noch einfacher als das Ändern. Dazu verwenden Sie die `Delete`-Methode der Zeile. Weil ich jetzt keines der wunderbaren, gerade erst hinzugefügten Bücher löschen will, füge ich mich als Autor an und lösche mich gleich wieder (in dem Umfeld habe ich nichts verloren ...):

```
authorTable.Rows.Add(new object[]
   {3, "Jürgen", "Bayer"});
authorTable.Rows[2].Delete();
```

Aktualisieren der Datenquelle und Fehlerauswertung

Wie Sie ja schon wissen, müssen Sie nach dem Anfügen, Aktualisieren oder Löschen von Datensätzen die `Update`-Methode des DataAdapter-Objekts aufrufen. Dabei können aber immer auch Fehler auftreten, besonders dann, wenn Sie die Regeln der Datenquelle nicht im `DataSet` abbilden (können). Wenn Sie beispielweise nur die Autoren unserer Buchverwaltung editieren, aber nicht die Bücher in das `DataSet` einlesen wollen (damit Sie diese mit den Autoren in Beziehung setzen können), kann der Anwender Autoren löschen, die eigentlich noch in der Büchertabelle referenziert werden. Beim Update wird dann eine Ausnahme erzeugt. Sie können dann entweder alle Änderungen über `RejectChanges` verwerfen oder besser alle Zeilen durchgehen, und über deren `HasErrors`-Eigenschaft ermitteln, ob die Zeile Fehler aufweist. Über die Eigenschaft `RowError` erhalten Sie die Fehlermeldung jeder Zeile. Über die `GetColumnError`-Methode können Sie sogar noch Fehler für einzelne Spalten ermitteln. Das ist ein deutliches Plus gegenüber anderen Technologien, die nur eine pauschale Fehlermeldung liefern. So können Sie sehr flexibel auf Fehler reagieren. Das folgende Beispiel regiert auf eine Ausnahme dadurch, dass der Anwender für jeden fehlerhaften Datensatz gefragt wird, ob er die Änderungen rückgängig machen will. Für die Anzeige des Datensat-

zes greife ich dabei auf die originalen Werte der Zeile zurück, da bei gelöschten Zeilen keine aktuellen Werte zur Verfügung stehen:

```
try
{
   dataAdapter.Update(dataSet, "Autoren");
}
catch (Exception ex)
{
    /* Zeilen durchgehen, um die fehlerhaften Zeilen
     * zu ermitteln */
   foreach (DataRow row in
      dataSet.Tables["Autoren"].Rows)
   {
      if (row.HasErrors)
      {
        string message = String.Format(
         "Fehler im Datensatz {0}: {1} {2}: {3}.\r\n" +
         "Wollen Sie den Datensatz wiederherstellen?",
         row["ID", DataRowVersion.Original],
         row["Vorname", DataRowVersion.Original],
         row["Nachname", DataRowVersion.Original],
         row.RowError);

         if (MessageBox.Show(message, "Fehler",
           MessageBoxButtons.YesNo,
           MessageBoxIcon.Error) == DialogResult.Yes)
         {
            /* Änderungen verwerfen */
            row.RejectChanges();
         }
      }
   }
}
```

Sie können das Ganze noch verfeinern, indem Sie den Status der Zeile abfragen und bei geänderten Datensätzen die aktuellen Werte anzeigen.

8.2.16 Sortieren und Filtern mit einem DataReader-Objekt

Wenn Sie Daten einer Datenquelle sortiert oder gefiltert ausgeben wollen, und dabei nur einmal sequenziell durch die Daten gehen wollen, verwenden Sie idealerweise ein DataReader-Objekt und spezifizieren im Command-Objekt einen entsprechenden Befehl. Bei relationalen Datenbanken, die mit SQL arbeiten, verwenden Sie dazu die `WHERE`- und die `ORDER BY`-Klausel. Das folgende Beispiel liest die Bücher des Autoren mit der ID 1 aus unserer Buchverwaltung sortiert nach dem Titel ein:

```
SqlConnection connection = new SqlConnection(
   @"Server=Zaphod\NetSDK;" +
   "Trusted_Connection=Yes;Database=Northwind");
connection.Open();

SqlCommand command = new SqlCommand("SELECT * FROM " +
   "Bücher WHERE AutorId = 1 ORDER BY Titel",
   connection);
SqlDataReader reader = command.ExecuteReader();
```

8.2.17 Das DataView-Objekt

Ein `DataSet`-Objekt können Sie natürlich genauso sortieren und filtern wie ein `DataReader`-Objekt. Die `DataSet`-Klasse stellt Ihnen aber auch die Möglichkeit zur Verfügung, ein `DataView`-Objekt aus den Daten zu erzeugen, über das Sie die in den einzelnen `DataTable`-Objekten gespeicherten Daten sehr flexibel sortieren, filtern, durchsuchen, durchgehen und editieren können. Ein `DataView`-Objekt stellt eine definierte Sicht auf eine Tabelle dar. Es kann an Steuerelemente gebunden werden. Sie können auf eine Tabelle mit mehreren `DataView`-Objekten unterschiedliche Sichten erzeugen. Die Sortierung und den Filter können Sie nach dem Erzeugen dynamisch einstellen:

```
SqlConnection connection = new SqlConnection(
   @"Server=Zaphod\NetSDK;" +
   "Trusted_Connection=Yes;Database=Northwind");
connection.Open();

/* DataSet erzeugen */
```

```
SqlDataAdapter adapter = new SqlDataAdapter(
   "SELECT * FROM Customers", connection);
DataSet dataSet = new DataSet();
adapter.Fill(dataSet, "Customers");

/* DataView auf der Kundentabelle erzeugen */
DataView dataView = new
   DataView(dataSet.Tables["Customers"]);
/* Sortierung einstellen */
dataView.Sort = "CompanyName";
/* Filtern */
dataView.RowFilter = "City = 'London'";

/* Durchgehen */
for (int i=0; i<dataView.Count; i++)
{
   Console.WriteLine(dataView[i]["CompanyName"]);
}
```

Die Ausdrücke für die Sortierung und den Filter müssen natürlich in der Befehlssprache der Datenquelle formuliert werden. Bei relationalen Datenbanken ist das der SQL-Dialekt der Datenbank. Die Sortierung und den Filter können Sie auch nachträglich umstellen:

```
dataView.RowFilter = "City = 'Berlin'";
for (int i=0; i<dataView.Count; i++)
{
   Console.WriteLine(dataView[i]["CompanyName"]);
}
```

Wenn Sie den Filter oder die Sortierung löschen wollen, setzen Sie die Eigenschaften auf einen Leerstring oder auf `null`.

Ein `DataView`-Objekt verwaltet ähnlich einem `DataTable`-Objekt einzelne Zeilen, die hier allerdings in Form von `DataRowView`-Objekten gespeichert sind. Über diese können Sie die Zeilen ansprechen. Sie können die Daten wie beim `DataTable`-Objekt auslesen, aktualisieren, anfügen und löschen.

Suchen können Sie über die `Find`-Methode in dem Feld, nach dem Sie sortiert haben. Diese Methode gibt den Index der Zeile zurück oder

einen Wert kleiner 0, wenn nichts gefunden wurde. Das folgende Beispiel sucht einen Kunden mit Namen »Simons bistro«:

```
int index = dataView.Find("Simons bistro");
if (index > -1)
   Console.WriteLine("Gefunden: {0}, {1}.",
      dataView[index]["CompanyName"],
      dataView[index]["City"]);
else
   Console.WriteLine("Nicht gefunden");
```

Wenn Sie mehrere Zeilen im Suchergebnis erwarten, können Sie die `FindRows`-Methode einsetzen, die ein Array aus `DataRowView`-Objekten mit den gefundenen Zeilen zurückgibt. Der folgende Quelltext ermittelt im `DataView` alle Kunden aus (good old) Germany. Dazu muss zunächst nach dem Feld *Country* sortiert werden:

```
dataView.Sort = "Country";
```

Dann können Sie suchen:

```
DataRowView[] rows = dataView.FindRows("Germany");
if (rows.Length > 0)
{
   Console.WriteLine("Gefunden:");
   for (int i=0; i<rows.Length; i++)
   {
      Console.WriteLine(rows[i]["CompanyName"]);
   }
}
else
   Console.WriteLine("Nicht gefunden");
```

Beachten Sie, dass Sie immer in den eventuell gefilterten Daten suchen.

Tabelle 8.10 zeigt schließlich noch die wichtigsten Eigenschaften und Methoden der `DataView`-Klasse.

Eigenschaft / Methode	Bedeutung
`AllowDelete`	Über diese Eigenschaft können Sie das Löschen von Daten unterbinden.
`AllowEdit`	Über diese Eigenschaft können Sie das Aktualisieren von Daten unterbinden.
`AllowNew`	Über diese Eigenschaft können Sie unterbinden, dass Datensätze angefügt werden.
`Count`	verwaltet die aktuelle Zahl der Datensätze, die dem Filter entsprechen
`DataRowView this[int recordIndex]`	Über den Indizierer erreichen Sie die einzelnen Zeilen.
`RowFilter`	definiert den Filter
`RowStateFilter`	Über diese Eigenschaft können Sie Zeilen filtern, die einen bestimmten Status besitzen. Sie können einen oder mehrere der Werte der Auflistung `DataViewRowState` übergeben: `Added` (neue Zeilen), `CurrentRows` (neue, modifizierte und unbearbeitet Zeilen), `Deleted` (gelöschte Zeilen), `ModifiedCurrent` (aktuelle Daten, die eine modifizierte Version der originalen Version sind), `ModifiedOriginal` (originale Daten, auch wenn die Zeile mittlerweile aktuelle Daten speichert), `None` (keine Daten), `OriginalRows` (originale Daten inklusive ungeänderten und gelöschten Zeilen) und `Unchanged` (ungeänderte Zeilen).
`Sort`	definiert die Sortierung
`Table`	spezifiziert die Tabelle

Eigenschaft / Methode	Bedeutung
`DataRowView AddNew()`	fügt eine neue Zeile an
`void Delete(int index)`	löscht eine Zeile
`int Find(...)`	sucht eine Zeile. Sie können einen einzelnen Wert oder ein Array aus Werten übergeben. Das `DataView`-Objekt muss nach dem Suchfeld sortiert sein.
`DataRowView[] FindRows(...)`	sucht mehrere Zeilen. Sie können einen einzelnen Wert oder ein Array aus Werten übergeben. Das `DataView`-Objekt muss nach dem Suchfeld sortiert sein.

Tabelle 8.10: Die wichtigsten Eigenschaften und Methoden des DataView-Objekts

9 Konfigurieren und Verteilen einer Anwendung

9.1 Anwendungskonfiguration

In der Praxis müssen viele Werte, mit denen eine Anwendung arbeitet, so verwaltet werden, dass der Anwender die Möglichkeit besitzt, diese zu ändern. Ein Beispiel dafür sind die einzelnen Argumente eines Verbindungsstring beim Datenzugriff. Früher – vor .NET – wurden solche Daten häufig in der Windows-Registrierdatenbank (englisch: Registry) oder in *.ini*-Dateien verwaltet. Die Registry ist unter .NET veraltet und verursacht in der Praxis viele Probleme, weil die Daten benutzerspezifisch gespeichert werden und weil Sie die Konfigurationsdaten nicht einfach mit Ihrer Anwendung ausliefern können. Ini-Dateien ermöglichen zwar die Auslieferung der Konfigurationsdaten mit der Anwendung, besitzen aber ein spezielles Format, das programmtechnisch nur mit speziellen (Windows-API-)Hilfsmitteln bearbeitet werden kann. Viel besser wäre, die Daten in einer XML-Datei zu verwalten. Und genau das stellt Ihnen .NET in Form von Anwendungs-Konfigurationsdateien zur Verfügung.

9.1.1 Die Basis: machine.config

Das .NET-Framework verwaltet eine Basis-Konfigurationsdatei mit dem Namen *machine.config* im Ordner *\WINNT\Microsoft.NET\Framework\vx.x.xxxx\CONFIG* (*x.x.xxxx* = aktuelle Versionsnummer des .NET-Frameworks). Die Konfigurationen, die in dieser Datei eingestellt sind, gelten zunächst für alle Anwendungen, die unter dem .NET-Framework ausgeführt werden. Die Datei liegt in reinem XML-Format vor und ist in UTF-8 codiert. Die Bearbeitung ist also nur mit einem Editor möglich, der diese Codierung beherrscht. Visual Studio ist dazu natürlich in der Lage.

Ich gehe hier nur auf die Grundlagen dieser Datei ein, weil die Einstellungen teilweise komplex sind, aber recht gut in Form von Kommentaren innerhalb der einzelnen Sektionen erläutert werden.

Die einzelnen Einstellungen werden in Sektionen verwaltet, die teilweise noch in Sektionsgruppen zusammengefasst sind. Im Element <*configSections*> werden diese zunächst beschrieben. Die Sektionsgruppe <*system.net*> beinhaltet beispielsweise Sektionen mit Einstellungen, die das Netzwerk betreffen. Die meisten Einstellungen innerhalb der Datei *machine.config* betreffen Webanwendungen. Wenn Sie Windows-Anwendungen entwickeln, können Sie die gesamte Sektionsgruppe <*system.web*> ignorieren. Eine für Windows-Anwendungen wichtige Einstellung ist die Einstellung *jitDebugging* im Element *system.windows.forms*, über die Sie das Debuggen bei unbehandelten Ausnahmen ein- oder ausschalten können.

Achten Sie bei der Bearbeitung der Datei darauf, dass Sie die korrekte Groß-/Kleinschreibung verwenden. XML unterscheidet zwischen Groß- und Kleinschreibung. Alle Elemente und Attribute innerhalb der Konfiguration sind über das camelCasing benannt.

9.1.2 Konfigurieren einer Anwendung

Sie können im Ordner jeder .NET-Anwendung eine Konfigurationsdatei ablegen, in der Sie die Einstellungen der *machine.config* für die Anwendung überschreiben und neue, anwendungsspezifische Einstellungen verwalten können. Diese Datei muss den kompletten Namen der Anwendung (inklusive der Endung!) mit der Endung *.config* besitzen. Die Konfigurationsdatei der Anwendung *Buchverwaltung.exe* heißt dann z. B. *Buchverwaltung.exe.config*.

Wollen Sie beispielsweise das JIT-Debuggen für eine Anwendung separat ein- oder ausschalten, tragen Sie das *system.windows.forms*-Element in die Anwendungs-Konfigurationsdatei ein:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
   <system.windows.forms jitDebugging="false" />
</configuration>
```

Beachten Sie, dass Sie das JIT-Debuggen nur für Windows-Anwendungen ausschalten können, nicht für Konsolenanwendungen.

Daneben sind noch eine Menge anderer vordefinierter Einstellungen möglich, die größtenteils nicht in der *machine.config* zu finden sind.

So können Sie beispielsweise für einzelne Assemblierungen, die Ihr Programm verwendet, in der Konfigurationsdatei definieren, welche Version und welche Kultur verwendet werden soll und wo die Assemblierung gespeichert ist. Suchen Sie im Index der .NET-Framework-Hilfe nach »configuration [.NET]«, um weitere Informationen zu erhalten.

Im Element *appSettings* können Sie eigene Konfigurationen ablegen, die Sie über *add*-Elemente hinzufügen. Sie erzeugen damit im Ergebnis eine Auflistung vom Typ `NameValueCollection`. Deshalb müssen Sie einen Schlüssel und einen Wert übergeben. Das folgende Beispiel definiert eigene Konfigurationseinträge für eine Datenbankverbindung:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
   <appSettings>
      <add key="server" value="(local)NetSDK" />
      <add key="trustedConnection" value="true" />
      <add key="userId" value="" />
    </appSettings>
</configuration>
```

Im Programm können Sie diese Einträge dann über die `AppSettings`-Eigenschaft der `ConfigurationSettings`-Klasse auslesen. Diese und andere Klassen zu Anwendungskonfiguration finden Sie im Namensraum `System.Configuration`.

```
string server =
   ConfigurationSettings.AppSettings["server"];
string userId =
   ConfigurationSettings.AppSettings["UserId"];
bool trustedConnection =
   ConfigurationSettings.AppSettings[
   "trustedConnection"] == "true" ? true : false;
```

Achten Sie darauf, dass diese Datei im Ordner der Anwendung gespeichert sein muss. Wenn Sie mit Visual Studio arbeiten, ist das der Ordner der aktuell verwendeten Build-Konfiguration (*debug*, *release*).

Diese einfache Konfiguration reicht für die Praxis häufig bereits aus. Sie müssen beim Lesen der Einstellungen natürlich darauf reagieren, dass ungültige Daten gespeichert sein können. Schön wäre auch noch, wenn geänderte Daten problemlos wieder zurückgeschrieben werden können. Das scheint aber mit den Mitteln des `System.Configuration`-Namensraums nicht zu gehen, wie Diskussionen in Newsgroups belegen. Sie können die XML-Datei aber über die Klassen des `System.XML`-Namensraums auch direkt bearbeiten. Auf der Buchseite bei *www.nitty-gritty.de* finden Sie einen Artikel dazu.

Programmdaten, die gelesen und geschrieben werden sollen, können Sie stattdessen auch in der Registry verwalten.

9.1.3 Registry-Zugriff

Über Klassen des Namensraums `Microsoft.Win32` können Sie die Registry lesen und beschreiben, um Konfigurations- oder Initialisierungsdaten Ihrer Anwendung zu verwalten. Das Lesen und Schreiben (!) ist einfach. Sie sollten aber beachten, dass Sie wichtige Registrydaten bei der Auslieferung der Anwendung mit ausliefern müssen. In einem Setup-Projekt können Sie die zu erzeugenden Registry-Einträge recht einfach definieren, wie ich noch im nächsten Abschnitt zeige.

Ein .NET-Programm kann jeden Schlüssel der Registry lesen und beschreiben, falls der Benutzer die erforderlichen Rechte besitzt. Ich gehe davon aus, dass Sie die Registry grundlegend kennen und beschreibe deswegen nur kurz, in welchen Schlüsseln der Registry Anwendungsdaten gespeichert werden:

- im Schlüssel *HKEY_LOCAL_MACHINE/Software*. Dann gelten diese Daten für alle Anwender, oder
- im Schlüssel *HKEY_CURRENT_USER/Software*. Dann gelten die Daten nur für den aktuell in Windows eingeloggten Anwender.

Beide Schlüssel besitzen Unterschlüssel, die den Hersteller der Software bezeichnen. Innerhalb dieser Schlüssel sind normalerweise wieder Unterschlüssel angelegt, die das jeweilige Produkt bezeichnen. Sie sollten sich an dieses Schema halten.

Wenn Sie Werte in der Registry verwalten wollen, benötigen Sie zunächst ein Objekt der Klasse `RegistryKey`, das einen Schlüssel repräsentiert. Eine Instanz dieser Klasse erhalten Sie über statische Eigenschaften der `Registry`-Klasse. Für unseren Zweck sind die Eigenschaften `CurrentUser` und `LocalMachine` interessant. Über die `OpenSubKey`-Methode können Sie Unterschlüssel öffnen. Mit `CreateSubKey` können Sie Unterschlüssel erzeugen. `CreateSubKey` ist so nett, einen bereits existierenden Schlüssel nicht zu überschreiben, sondern einfach zum Schreiben zu öffnen. Beide Methoden geben ein `RegistryKey`-Objekt zurück. Über die `GetValue`-Methode können Sie schließlich Daten lesen, mit `SetValue` schreiben Sie Werte. Beim Lesen müssen Sie beachten, dass die Methoden `null` zurückgeben, falls ein Schlüssel bzw. Wert nicht existiert. Das folgende Beispiel liest die Konfigurationsdaten *Server* und *User-Id* aus dem Schlüssel *HKEY_LOCAL_MACHINE/Software/Addison-Wesley/RegistryDemo*:

```
/* Variablen für die Daten */
string server = null, userId = null;

/* Schlüssel zum Lesen öffnen */
RegistryKey key = Registry.LocalMachine.
   OpenSubKey("Software").
   OpenSubKey("Addison-Wesley").
   OpenSubKey("RegistryDemo");
/* Werte lesen, falls diese vorhanden sind */
if (key != null)
{
   object keyValue = key.GetValue("Server");
   if (keyValue != null)
      server = keyValue.ToString();
   keyValue = key.GetValue("UserId");
   if (keyValue != null)
      userId = keyValue.ToString();
   /* Schlüssel schließen */
   key.Close();
}
```

Der folgende Quellcode schreibt die Daten in die Registry:

```
/* Die Daten */
string server = "Zaphod";
string userId = "Ford";

/* Schlüssel 'HKEY_LOCAL_MACHINE/Software'
 * zum Schreiben öffnen */
RegistryKey softwareKey =
   Registry.LocalMachine.OpenSubKey("Software", true);
/* Unterschlüssel 'Addison-Wesley' anfügen
 * bzw. öffnen */
RegistryKey companyKey =
   softwareKey.CreateSubKey("Addison-Wesley");
/* Unterschlüssel 'RegistryDemo' anfügen. */
RegistryKey productKey =
   companyKey.CreateSubKey("RegistryDemo");
/* Werte schreiben */
productKey.SetValue("Server", server);
productKey.SetValue("UserId", userId);
/* Werte in die Registry übertragen */
companyKey.Flush();
/* Schlüssel schließen */
companyKey.Close();
productKey.Close();
softwareKey.Close();
```


9.2 Verteilen einer Anwendung

9.2.1 Copy & Paste

Wenn Ihre Anwendung keine speziellen Assemblierungen nutzt, die im globalen Assemblierungs-Cache verwaltet oder über die Konfigurationsdatei der Anwendung eingebunden werden, ist das Verteilen einer Anwendungen ganz einfach: Ein simples Kopieren der Dateien, die Sie im *release*-Ordner finden, reicht aus, um eine Anwendung auf einem anderen Computer auszuführen. Alle referenzierten externen Assemblierungen befinden sich automatisch nach dem Kompilieren in diesem Ordner.

Auf dem anderen Computer muss natürlich das .NET-Framework installiert sein, damit Ihre Anwendung ausgeführt werden kann. Eine Frage, die zurzeit noch nicht beantwortet werden kann, ist, ob eine .NET-Anwendung auch ausgeführt wird, wenn auf dem Zielrechner eine *neuere* Version des .NET-Framework installiert ist als die, mit der Sie die Anwendung kompiliert haben. Theoretisch sollte das der Fall sein. Ob das in der Praxis auch funktioniert, bleibt abzuwarten. Es wäre auf jeden Fall sinnvoll, die Version des .NET-Frameworks mitzuliefern, mit der Sie die Anwendung kompiliert haben. Liefern Sie aber vielleicht nur das .NET-Framework in der verteilbaren (Redist-)Version aus. Diese Version enthält keine SDK-Tools und ist wesentlich kleiner als das NET-Framework SDK.

Webbasierte Anwendungen können Sie mit Hilfe des Internet Explorers ab Version 5.5 auch über das Internet verteilen. Windows-Anwendungen oder Komponenten können Sie über Windows Installer-Pakete verteilen. Der Windows Installer kann Assemblierungen im globalen Assemblierungen-Cache installieren und erlaubt die Deinstallation einer Anwendung. Wenn Sie den Windows Installer nicht direkt konfigurieren möchten, können Sie dazu einfach ein Visual Studio Setup-Projekt verwenden.

9.2.2 Visual Studio Setup-Projekte

Um ein Windows Installer Setup mit Visual Studio zu erzeugen, integrieren Sie ein Setup-Projekt in die Projektmappe der Anwendung. Öffnen Sie also die Projektmappe Ihrer Anwendung und fügen Sie dieser über das Menü DATEI / PROJEKT HINZUFÜGEN ein neues Projekt hinzu. Wählen Sie im Projektauswahldialog den Projekttyp SETUP- UND WEITERGABEPROJEKTE und am besten den Eintrag SETUP-ASSISTENT. Wenn Sie dann den OK-Schalter betätigt haben, erscheint der Setup-Assistent. In Schritt 1 werden lediglich einige Informationen ausgegeben. Die wichtigste Information in diesem Schritt ist, dass Sie auf WEITER klicken sollten, was Sie dann auch tun ...

Im zweiten Schritt des Setup-Assistenten können Sie die Art des zu erzeugenden Setups auswählen (Abbildung 9.1).

Lassen Sie die erste Option für die Verteilung einer Windows-Anwendung eingeschaltet.

Bild 9.1: Schritt 2 des Setup-Assistenten

Im nächsten Schritt wählen Sie die Dateigruppen aus, die in das Setup integriert werden sollen (Abbildung 9.2).

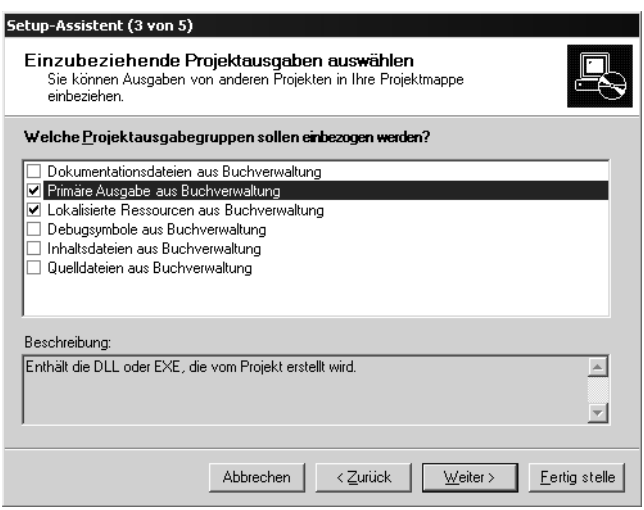

Bild 9.2: Schritt 3 des Setup-Assistenten

Wichtig ist die zweite Option in Abbildung 9.2, die die EXE-Datei und alle referenzierten, externen Assemblierungen beinhaltet. Der Setup-Assistent erkennt alle direkten Abhängigkeiten Ihrer Anwendung und integriert die referenzierten Assemblierungen automatisch in das Setup. Die dritte Option müssen Sie nur dann einschalten, wenn Sie verschiedene Ressourcen-Assemblierungen für unterschiedliche Kulturen mitliefern. Die weiteren Optionen sind für die Installation auf normalen Anwender-Computern nicht interessant. Diese Optionen benötigen Sie, wenn Sie Ihr Programm inklusive Quellcode und Dokumentation einem anderen Programmierer übergeben wollen (das geht zwar auch durch einfaches Kopieren, in einem Setup sind allerdings alle referenzierten externen Assemblierungen enthalten).

Im nächsten Schritt können Sie zusätzliche Dateien in das Setup übernehmen. Da der Assistent die Konfigurationsdatei einer Anwendung nicht automatisch integriert, müssen Sie zumindest diese selbst in das Setup integrieren.

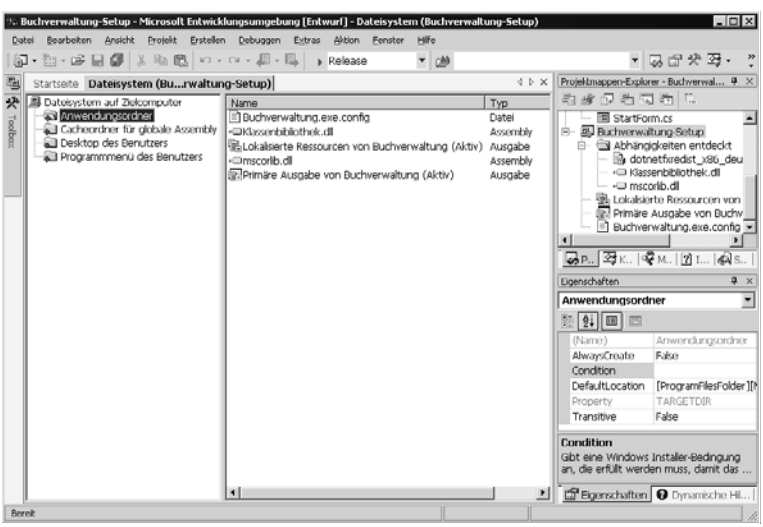

Bild 9.3: Das Setup-Projekt

Nachdem Sie dann den nächsten Schritt (der nur einige Informationen ausgibt) bestätigt haben, ist das Projekt zunächst fertiggestellt. Im

Projekt können Sie noch eine Menge Einstellungen vornehmen. Wichtig sind zunächst die Projekteigenschaften selbst. Achten Sie besonders auf den Produktnamen und den Titel des Projekts. Der Produktname wird per Default beim Setup als Zielordner vorgeschlagen.

Das Setup-Projekt stellt in einem Dateisystem-Editor verschiedene Ordner und die enthaltenen Daten des Zielrechners dar. Sie können (über das Kontextmenü des Ordners) Dateien, Verknüpfungen und Unterordner in diese Ordner integrieren. Das Setup wird die von Ihnen definierte Struktur auf dem Zielrechner anlegen.

Falls der Dateisystem-Designer einmal aus Versehen geschlossen wurde, können Sie diesen über das Kontextmenü des Projekteintrags im Projektmappen-Explorer über den Befehl Ansicht / Dateisystem *wieder öffnen.*

Sinnvoll ist die Definition von Verknüpfungen zur Programmdatei im Desktop- und im Programm-Menü-Ordner. Wählen Sie dazu den Befehl Neue Verknüpfung erstellen im Kontextmenü eines Ordners. Führen Sie im Dialog zum Hinzufügen von Elementen (Abbildung 9.4) einen Doppelklick auf den Anwendungsordner aus und wählen Sie den Eintrag Primäre Ausgabe von ...

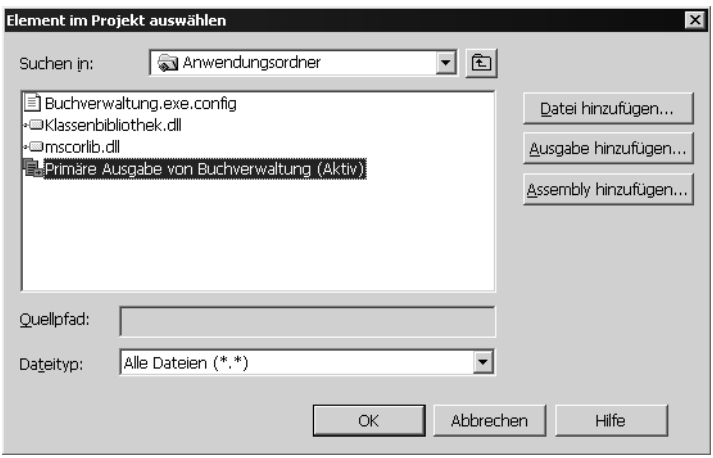

Bild 9.4: Hinzufügen einer Verknüpfung zu der Exe-Datei der Anwendung zum Programmmenü des Zielrechners

Die primäre Ausgabe des Projekts ist bei einer Windows-Anwendung die Exe-Datei.

Über den Ordner CACHEORDNER FÜR GLOBALE ASSEMBLIERUNGEN können Sie die externen Assemblierungen, die Ihr Programm referenziert, in den globalen Assemblierungs-Cache (GAC) des Zielrechners installieren. Zuvor sollten Sie sich aber erst noch ein wenig mit der Theorie des GAC auseinandersetzen. Assemblierungen im GAC müssen beispielsweise einen »strengen Namen« (englisch: Strong Name) besitzen (der u. a. einen eindeutigen Schlüssel enthält), damit sie installiert werden können.

Im Kontextmenü des linken Fensters der Ordneranzeige können Sie noch spezielle Ordner, wie den *System*-Ordner oder den *Send To*-Ordner hinzufügen, um dort ebenfalls Dateien oder Verknüpfungen zu installieren.

Für den Programmordner können Sie den Zielordner einstellen, der dem Benutzer beim Setup vorgeschlagen wird. Im Eigenschaftenfenster finden Sie dazu die Eigenschaft `DefaultLocation`. Hier können Sie vordefinierte Makros verwenden. *[ProgramFilesFolder]* steht beispielsweise für den System-Programmordner, *[Manufacturer]* für den Text, den Sie in der gleichnamigen Eigenschaft des Projekts eingegeben haben und *[ProductName]* für den ebenfalls dort verwalteten Produktnamen. Zusätzlich können Sie noch für die Ordner, Dateien und Verknüpfungen Bedingungen definieren, die dann dazu führen, dass das jeweilige Element nur dann installiert wird, wenn die Bedingung erfüllt ist.

Eine weitere Möglichkeit, die das Setup-Projekt bietet, ist das Erzeugen von Registry-Einträgen. Wählen Sie dazu den Befehl ANSICHT / REGISTRIERUNG im Kontextmenü des Projekteintrags (im Projektmappen-Explorer). Falls Ihre Anwendung mit Registrydaten arbeitet, können Sie diese im Registrierungs-Designer definieren. Das Setup schreibt diese Daten dann automatisch in die Registry.

Die Besprechung der weiteren Möglichkeiten des Setup-Projekts würde hier den Rahmen sprengen. In der Hilfe (F1 bei aktivem Dateisystem-Editor) sind die Möglichkeiten recht gut beschrieben.

Um das Setup zu erzeugen, erstellen Sie die Exe-Datei des Setup-Projekts (nicht die der Projektmappe!). Visual Studio erzeugt auch die Exe-Datei der Anwendung neu, bevor die Exe-Datei des Setup-Projekts erzeugt wird. Wenn kein Fehler aufgetreten ist, können Sie die Setup-Dateien, die Sie im *Release*-Ordner des Setup-Projekts finden, ausliefern.

Stichwortverzeichnis

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z