# C# lernen

## Die Lernen-Reihe

In der Lernen-Reihe des Addison-Wesley Verlages sind die folgenden Titel bereits erschienen bzw. in Vorbereitung:

André Willms
C-Programmierung lernen
432 Seiten, ISBN 3-8273-1405-4

André Willms
C++-Programmierung lernen
408 Seiten, ISBN 3-8273-1342-2

Guido Lang, Andreas Bohne
Delphi 5 lernen
432 Seiten, ISBN 3-8273-1571-9

Walter Hergoltz
HTML lernen
323 Seiten, ISBN 3-8273-1717-7

Judy Bishop
Java lernen
636 Seiten, ISBN 3-8273-1605-7

Michael Schilli
Perl 5 lernen
ca. 400 Seiten, ISBN 3-8273-1650-9

Michael Ebner
SQL lernen
336 Seiten, ISBN 3-8273-1515-8

René Martin
VBA mit Word 2000 lernen
412 Seiten, ISBN 3-8273-1550-6

René Martin
VBA mit Office 2000 lernen
576 Seiten, ISBN 3-8273-1549-2

Patrizia Sabrina Prudenzi
VBA mit Excel 2000 lernen
512 Seiten, ISBN 3-8273-1572-7

Patrizia Sabrina Prudenzi, Dirk Walter
VBA mit Access 2000 lernen
680 Seiten, ISBN 3-8273-1573-5

Dirk Abels
Visual Basic 6 lernen
425 Seiten, ISBN 3-8273-1371-6

**Frank Eller**

# C# lernen

## anfangen, anwenden, verstehen


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.**

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Fast alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden, sind gleichzeitig auch eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Umwelthinweis:
Dieses Produkt 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

04 03 02 01

ISBN 3-8273-1784-3

© 2001 by Addison Wesley Verlag,
ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany
Alle Rechte vorbehalten

| | |
|---|---|
| Einbandgestaltung: | Barbara Thoben, Köln |
| Lektorat: | Christina Gibbs, cgibbs@pearson.de |
| Korrektorat: | Simone Burst, Großberghofen |
| Herstellung: | Ulrike Hempel, uhempel@pearson.de |
| Satz: | mediaService, Siegen |
| Druck und Verarbeitung: | Bercker, Kevelaer |

Printed in Germany

# Inhaltsverzeichnis

# V Vorwort

Wenn ein Verlag Ihnen anbietet, ein Buch über eine Programmiersprache zu verfassen, ist das natürlich eine schöne Sache. Wenn es sich dabei noch um eine von Grund auf neue Programmiersprache handelt, ist die Freude darüber nochmals größer, denn welcher Autor kann schon von sich behaupten, eines der ersten Bücher über eine Programmiersprache geschrieben zu haben?

Es ist aber auch eine besondere Herausforderung. Nicht nur, dass es sich um eine neue Programmiersprache handelt, auch das Konzept des Buches musste erarbeitet werden. Visual Studio.net war noch nicht verfügbar, nicht einmal als Beta, und der Zeitpunkt, zu dem Microsoft die erste Version veröffentlichen würde, war vollkommen unbekannt.

Ich entschied mich daher, die Entwicklungssoftware außer Acht zu lassen und konzentrierte mich auf die Sprache selbst. Dabei habe ich versucht, auch für den absoluten Neueinsteiger, der noch nie mit der Programmierung von Computern zu tun hatte, verständlich zu bleiben. Obwohl das immer eine Gratwanderung ist, gerade bei Büchern für Einsteiger, hoffe ich doch, viel Information in verständlicher Art zusammengestellt zu haben. Dabei ist klar, dass dieses Buch nur die Basis darstellt; ein Buch kann Ihnen immer nur eine gewisse Menge an Kenntnissen vermitteln, das weitaus größeren Wissen erhalten Sie, wenn Sie mit der Programmiersprache arbeiten.

Manche werden in diesem Buch Details über die Programmierung von Benutzerschnittstellen, von Windows-Oberflächen oder Internet-Komponenten vermissen. Um diese Bestandteile aber effektiv programmieren zu können, ist es nötig, eine Entwicklungssoftware wie das Visual Studio zu verwenden. Natürlich ist es möglich, auch rein textbasiert alle Bestandteile eines Windows-Programms zu erzeugen, jedoch würden diese Vorgehensweisen nicht nur den Rahmen

des Buches sprengen, sondern einen Einsteiger doch etwas überfordern. Wie bereits gesagt, jedes Buch ist eine Gratwanderung.

Ich hoffe, dass der Inhalt dennoch nützlich für Sie ist. An dieser Stelle noch mein Dank an die Mitarbeiter des Verlags Addison-Wesley und ganz besonders an meine Lektorin Christina Gibbs, die mich immer mit allen notwendigen Unterlagen versorgt hat und stets ein kompetenter Ansprechpartner war.

*Frank Eller*

*Januar 2001*

# 1 Einführung

Willkommen im Buch C# lernen.

C# (gesprochen „C Sharp“) ist eine neue, objektorientierte Programmiersprache, die von Microsoft unter anderem zu dem Zweck entworfen wurde, das Programmieren zu vereinfachen ohne die Möglichkeiten, die heutige Programmiersprachen bieten, einzuschränken. Hierzu wurde nicht nur eine neue Programmiersprache entwickelt, sondern ein komplettes Konzept. Der Name dieses Konzepts ist *.net* (gesprochen: dotnet).

Das Ziel, das Microsoft damit verfolgt, ist eine noch größere Integration des Internets mit dem Betriebssystem. Das Internet soll in der Zukunft die Plattform für alle Anwendungen werden, und C# als Programmiersprache ist die erste Sprache die dies voll unterstützt. Dabei basiert C# auf einer Laufzeitumgebung, die die Basis für das neue Konzept bildet. Zunächst vorgestellt als *Next Generation Windows Services* (auf deutsch etwa „Windows-Dienste der nächsten Generation“) trägt sie heute den Namen *.net-Framework*. Der Name legt bereits die Vermutung nahe, dass Microsoft es auf die Verschmelzung von Internet und Betriebssystem abgesehen hat. Tatsächlich ist es so, dass das Internet als Plattform für Anwendungen bzw. Dienste etabliert werden soll. Ob es sich bei der Verschmelzung lediglich um Windows und das Internet handelt oder ob das .net-Framework auch noch für andere Betriebssysteme erhältlich sein wird, ist derzeit noch nicht bekannt. Das Konzept jedoch ist interessant und viel versprechend; wir werden daher die Hintergründe von .net im Laufe dieser Einführung noch kurz beleuchten.

## 1.1 Anforderungen ...

### 1.1.1 ... an den Leser ...

Das vorliegende Buch soll Ihnen dabei helfen, die Sprache C# zu erlernen. Es ist ein Buch für Einsteiger in die Programmierung oder auch für Umsteiger, die bereits in einer oder mehreren Programmiersprachen Erfahrung gesammelt haben. Die Voraussetzungen, die das Buch an den Leser stellt, sind daher eigentlich sehr niedrig. Sie sollten auf jeden Fall mit dem Betriebssystem Windows umgehen können. Außerdem sollten Sie ein wenig logisches Denken mitbringen, ansonsten benötigen Sie keine weiteren Kenntnisse. Programmiererfahrung in einer anderen Programmiersprache als C# ist zwar von Vorteil, aber im Prinzip nicht notwendig.

### 1.1.2 ... und an den Computer

Die Anforderungen an das System sind leider etwas höher als die an den Leser. Sie benötigen einen Rechner, der eine gewisse Geschwindigkeit und ein gewisses Maß an Hauptspeicher mitbringt.

- Mindestens Pentium II mit 166 MHz, empfohlen Pentium III mit mindestens 500 MHz. Wahlweise auch AMD Athlon oder K6-2.
- Betriebssystem Windows 98, Me, NT4 oder 2000 mit installiertem Internet Explorer 5.5. Unter Windows 95 läuft das .net-Framework noch nicht, und es ist fraglich ob Microsoft es noch dafür bereitstellen wird.
- Für die Internet-Funktionalität installierter Internet Information Server. Dieser befindet sich auf der CD Ihres Betriebssystems oder Sie finden ihn im Internet unter *www.microsoft.com*.
- Microsoft Data Access Components (MDAC) in der Version 2.6.
- 128 MB Ram
- Mindestens 250 MB Festplattenspeicher während der Installation des .net-Frameworks, nach der Installation belegt die Software noch ungefähr 150 MB.
- Die heutigen Computer sind fast alle schon ab Werk mit 128 MB und einer verhältnismäßig großen Festplatte (unter 20 GB läuft da nichts mehr) ausgerüstet. In Hinsicht auf den Speicher sollte es also keine Probleme geben. Sowieso sollten Programmierer sich stets mit einem relativ schnellen und gut ausgebauten System ausstatten.

Oftmals ist es sinnvoll, als Programmierer mehrere Betriebssysteme installiert zu haben. Eines für die tägliche Arbeit und eines zum Testen, quasi als „Programmierumgebung“. Bei der Programmierung ist es immer möglich, dass ein System mal „zerschossen“ wird und komplett neu installiert werden muss. Aber auch das ist bei heutigen Computern und Betriebssystemen kein Problem mehr.

## 1.2 Das Buch

### 1.2.1 Schreibkonventionen

Wie für alle Fachbücher, so gelten auch für dieses Buch diverse Schreibkonventionen, die die Übersicht innerhalb der Kapitel, des Fließtextes und der Quelltexte erhöhen. In der Tabelle 1.1 finden Sie die in diesem Buch benutzten Formatierungen und für welchen Zweck sie eingesetzt werden.

| Formatierung | Einsatzzweck |
|---|---|
| *kursiv* | Wichtige Begriffe werden in kursiver Schrift gesetzt. Normalerweise werden diese Begriffe auch im gleichen Absatz erklärt. Falls es sich um englische Begriffe oder Abkürzungen handelt, steht in Klammern dahinter der vollständige Ausdruck bzw. auch die deutsche Übersetzung. |
| `fest` | Der Zeichensatz mit festem Zeichenabstand wird für Quelltexte im Buch benutzt. Alles, was irgendwie einen Bezug zu Quelltext hat, wie Methodenbezeichner, Variablenbezeichner, reservierte Wörter usw., wird in dieser Schrift dargestellt. |
| Tastenkappen | Wenn es erforderlich ist, eine bestimmte Taste oder Tastenkombination zu drücken, wird diese in Tastenkappen dargestellt, z.B. Alt + F4. |
| **fett** | Fett gedruckt werden reservierte Wörter im Quelltext, zur besseren Übersicht. |
| KAPITÄLCHEN | Kapitälchen werden verwendet für Programmnamen, Menüpunkte und Verzeichnisangaben, außerdem für die Angaben der Schlüssel in der Registry. |

*Tabelle 1.1: Die im Buch verwendeten Formatierungen*

### 1.2.2 Syntaxschreibweise

Die Erklärung der Syntax einer Anweisung erfolgt ebenfalls nach einem vorgegebenen Schema. Auch wenn es manchmal etwas kompliziert erscheint, können Sie doch die verschiedenen Bestandteile eines Befehls bereits an diesem Schema deutlich erkennen.

– Optionale Bestandteile der Syntax werden in eckigen Klammern angegeben, z.B. [Modifikator]
– Reservierte Wörter innerhalb der Syntax werden fett gedruckt, z.B. **class**.
– Alle anderen Bestandteile werden normal angegeben.
– Achten Sie darauf, dass auch die Sonderzeichen wie z.B. runde oder geschweifte Klammern zur Syntax gehören.
– Es wird Ihnen nach einer kurzen Eingewöhnungszeit nicht mehr schwer fallen, die genaue Syntax eines Befehls bereits anhand des allgemeinen Schemas zu erkennen. Außerdem wird dieses Schema auch in den Hilfedateien des Visual Studio bzw. in der MSDN-Library verwendet, so dass Sie sich auch dort schnell zurechtfinden werden.

### 1.2.3 Symbole (Icons)

Sie werden im Buch immer wieder Symbole am Rand bemerken, die Sie auf etwas hinweisen sollen. Die folgenden Symbole werden im Buch verwendet:

Dieses Symbol steht dort, wo es etwas **Wichtiges** zu beachten gibt, sei es nun bei der Programmierung oder der Verwendung eines Feature der Programmiersprache. Sie sollten sich diese Abschnitte immer durchlesen.

Dieses Symbol steht für einen **Hinweis**, der möglicherweise bereits aus dem Kontext des Buchtextes heraus klar ist, aber nochmals erwähnt wird. Nicht immer nimmt man alles auf, was man liest. Die Hinweise enthalten ebenfalls nützliche Informationen.

Dieses Symbol steht für einen **Tipp** des Autors an Sie, den Leser. Zwar sind Autoren auch nur Menschen, aber wir haben doch bereits gewisse Erfahrungen gesammelt. Tipps können Ihnen helfen, schneller zum Ziel zu kommen oder Gefahren zu umschiffen.

Hinter diesem Symbol verbirgt sich ein **Beispiel**, also Quelltext. Beispiele sind eine gute Möglichkeit, Ihr Wissen zu vertiefen. Sehen Sie sich die Beispiele des Buchs gut an, und Sie werden recht schnell ein Verständnis für die Art und Weise bekommen, wie die Programmierung mit C# vor sich geht.

Dieses Symbol steht für **Übungen**, die Sie durchführen sollen. Alle diese Übungen haben ebenfalls den Sinn und Zweck, bereits gelernte Begriffe und Vorgehensweisen zu vertiefen, und verschaffen Ihnen so ein größeres Wissen bzw. festigen Ihr Wissen dadurch, dass Sie es anwenden.

### 1.2.4 Aufbau

Das Buch erklärt die Programmiersprache C# anhand einer großen Anzahl von Beispielen. Es geht vor allem darum, Syntax und grundsätzliche Möglichkeiten der Sprache klarzumachen, nicht um die Programmierung umfangreicher Applikationen. Auf besonders tief schürfende Erklärungen wird daher verzichtet, stattdessen erhalten Sie einen Überblick über die Sprache selbst und ihre Eigenheiten. Einige Erklärungen sind natürlich unumgänglich, will man eine neue Programmiersprache erlernen. Ebenso kann es vorkommen, dass in manchen Kapiteln Dinge vorkommen, die erst in einem späteren Teil des Buches erklärt werden. Das lässt sich aufgrund der Komplexität und Leistungsfähigkeit heutiger Programmiersprachen leider nicht vermeiden.

*Referenzen*

Während die einzelnen Kapitel Ihnen Stück für Stück die Programmiersprache erklären, finden Sie im Anhang einige Referenzen, die bei der täglichen Arbeit nützlich sind. Unter anderem werden wichtige Tabellen, die auch in den Kapiteln angegeben sind, dort nochmals wiederholt. Damit haben Sie mit einem Griff alle Informationen, die Sie benötigen, und müssen nicht mühevoll im gesamten Buch nachschlagen, nur weil Sie z.B. eine Auflistung der reservierten Wörter von C# suchen.

## 1.3 Das .net-Framework

### 1.3.1 Die Installation des .net-Frameworks

Bevor es an die Installation des SDK geht, sollten Sie die Datenzugriffskomponenten und den Internet Explorer 5.5 installiert haben. Beides wollte ich Ihnen eigentlich auf der Buch-CD zur Verfügung stellen, wofür ich aber die Zustimmung von Microsoft benötigt hätte. Die sind in der Hinsicht leider immer etwas zugeknöpft, daher muss ich Sie leider auf den Download von der Microsoft-Website verweisen.

Die Installation ist relativ problemlos, starten Sie einfach das Programm SETUP.EXE und bestätigen Sie die Fragen, die Ihnen gestellt werden (wie immer Zustimmung zum Lizenzabkommen usw.). Danach wird das SDK in das Verzeichnis C:\PROGRAMME\MICROSOFT.NET\FRAMEWORKSDK installiert.

Nach der Installation ist evtl. ein Neustart notwendig. Danach können Sie mit dem Programmieren in einer neuen, faszinierenden Sprache beginnen.

### 1.3.2 Installation mit Visual Studio .net Beta

Im Internet ist mittlerweile auch die erste Beta der neuen Version von Visual Studio erhältlich. Für Abonnenten des Microsoft Developer Network (MSDN) ist der direkte Download von der Microsoft Website möglich, alle anderen können die CD ordern.

Alle benötigten Erweiterungen für die unterstützten Betriebssysteme sind auf CD enthalten. Der Wermutstropfen ist, dass die Beta verständlicherweise nur in englischer Sprache erhältlich ist. Das ist zumindest bei einer Installation unter Windows 2000 ein Nachteil, da hier das Service Pack 1 erforderlich ist – in Landessprache. Der Download von der Microsoft-Website ist fast 19 MB dick, auch mit Kanalbündelung noch ein Download von nahezu 30 Minuten. Unter Windows ME installierte sich das Paket dafür ohne Probleme.

Das Design des Installationsprogramms zeigt bereits das Vorhaben Microsofts, das Internet noch mehr in die Anwendungsentwicklung mit einzubeziehen. Vollkommen neu gestaltet zeigt es sich in ansprechendem Web-Design. Die Installation selbst funktioniert reibungslos und fast ohne Eingriffe des Anwenders.

### 1.3.3 Einige Grundlagen über .net

Der frühere Name des .net-Frameworks war *NGWS* (Next Generation Windows Services, Windows-Dienste der nächsten Generation). Der endgültige Name ist allerdings *.net-Framework*, ebenso wie die Bezeichnung für die nächste Entwicklungssoftware von Microsoft nicht „Visual Studio 7“ sein wird, sondern „Visual Studio .net“. Der Grund für diese drastische Änderung ist die neue Konzeption der Programmiersprachen, die Microsoft mit dem .net-Framework betreiben will.

Das .net-Framework ist eine Laufzeitumgebung für mehrere Programmiersprachen. Im Klartext bedeutet dies, dass Programme, die auf .net aufbauen (wie z.B. alle mit C# geschriebenen Programme) diese Laufzeitumgebung benötigen, um zu laufen. Möglicherweise kennen Sie das noch von Visual Basic, wo ja immer die berühmt-berüchtigten *VBRunXXX.dll*-Dateien mitgeliefert werden mussten. Auch diese DLLs nannte man *Laufzeit-DLLs*.

Das .net-Framework ist aber noch mehr. Nicht nur, dass es mehrere Programmiersprachen unterstützt und dass auch weitere Programmiersprachen für eine Unterstützung des .net-Frameworks angepasst werden können, es enthält auch eine Sprachspezifikation und eine Klassenbibliothek, die ganz auf dem neuen Konzept aufgebaut ist. Mit Hilfe dieser Spezifikationen will Microsoft einiges erreichen:

– Vereinfachung der **Programmierung** ohne Verlust von Möglichkeiten.
– Sprachen- und Systemunabhängigkeit, da Programme auf jedem Betriebssystem laufen, das .net enthält.
– Eine bessere Versionskontrolle und damit eine bessere Übersichtlichkeit über die im System installierten Dlls.
– Sprachenübergreifende Komponenten, die nicht nur von jeder Sprache, die .net unterstützt, benutzt, sondern auch erweitert werden können.
– Sprachenübergreifende Ausnahmebehandlung (Ausnahmen sind Fehler, die zur Laufzeit des Programms auftreten können, z.B. Division durch Null); d.h. ein einheitliches System zum Abfangen und Behandeln von Fehlern zur Laufzeit.

Erreicht wird dies dadurch, dass die verschiedenen Programmiersprachen, die .net nutzen wollen, sich den darin festgelegten Vorgaben unterordnen müssen. Bei diesen Vorgaben handelt es sich um den kleinsten gemeinsamen Nenner zwischen den einzelnen Programmiersprachen. Die Bezeichnung Microsofts für diese Spezifikation ist *CLS* (Common Language Spezification, Allgemeine Sprachspezifikation). Entsprechend ist die Bezeichnung für die sprachenübergreifende Laufzeitumgebung *CLR* (Common Language Runtime, Allgemeine Laufzeitumgebung).

Weiterhin enthält das .net-Framework ein virtuelles Objektsystem, d.h. es stellt Klassen zur Verfügung, die vereinheitlicht sind und von jeder Programmiersprache (die .net unterstützt) benutzt werden können. C# als Systemsprache von .net macht intensiven Gebrauch von dieser Klassenbibliothek, da es selbst keine beinhaltet. Das bedeutet, C# nutzt .net am intensivsten aus. Es ist eine Sprache, die speziell für dieses Konzept zusammengebaut wurde. Für den Erfolg, also dafür, dass es wirklich eine gute, übersichtliche und verhältnismäßig leicht zu erlernende Sprache ist, zeichnet Anders Heijlsberg verantwortlich, der unter anderem auch bei der Entwicklung von Borland Delphi mit im Spiel war.

***Features von .net***

Die Vorteile sowohl des virtuellen Objektsystems als auch der allgemeinen Sprachspezifikation sind, dass das .net-Framework selbst die Kontrolle über die Vorgänge zur Laufzeit übernehmen kann, während früher der Programmierer selbst dafür verantwortlich war. Damit ergeben sich einige Möglichkeiten, die vor allem für die Sprache C# gelten, aber wie gesagt auch von anderen Sprachen implementiert werden können:

- *managed Code* (verwalteter Code). Bisher musste der Programmierer sich um alles kümmern, was in seinem Code ablief. Nun übernimmt das .net-Framework eine große Anzahl der zeitaufwändigen Arbeiten selbst und entlastet damit den Programmierer.
- *Garbage-Collection* (sozusagen eine Speicher-Müllabfuhr). Bisher musste sich der Programmierer darum kümmern, reservierten Speicher wieder an das System zurückzugeben. C/C++ Programmierer werden wissen, wovon ich spreche. Alle Objekte, für die Speicher reserviert wurde, mussten auch wieder freigegeben werden, weil ansonsten so genannte „Speicherleichen" zurückblieben, also Speicher auch nach Programmende noch belegt blieb. Das .net-Framework beinhaltet für managed Code (also Code, der sich an die CLS hält) eine automatische Garbage-Collection, d.h. Objekte, die nicht mehr referenziert werden, werden automatisch und sofort aus dem Speicher entfernt und dieser wieder freigegeben. Das Ergebnis: Schluss mit den Speicherleichen und eine Vereinfachung der Programmierung.
- *Systemunabhängigkeit*, da keine Systemroutinen mehr direkt aufgerufen werden, wenn mit managed Code gearbeitet wird. Das .net-Framework liegt zwischen Programm und Betriebssystem, was bedeutet, dass Aufrufe von Funktionen auf jedem Betriebssystem, das .net unterstützt, genau den gewünschten Effekt haben.

An dieser Stelle soll dies an Vorteilen genügen. Für Sie als C#-Programmierer ist das .net-Framework aus mehreren Gründen wichtig. Zum einen stellt es die Laufzeitumgebung für alle C#-Programme dar, d.h. ohne .net läuft keine C#-Applikation. Zum Zweiten enthält es die Klassenbibliothek, also die vorgefertigten Klassen, Datentypen und Funktionen für C#. Die Sprache selbst enthält keine Klassenbibliothek, sondern arbeitet ausschließlich mit der des .net-Frameworks. Der dritte Grund sind die Compiler, die im Framework mit enthalten sind und ohne die Sie ebenfalls kein lauffähiges Programm erstellen können. Und da Visual C++ bzw. Visual Basic ebenfalls gleich an .net angepasst wurden, enthält das Framework auch deren Compiler.

Sie werden sich fragen, warum Sie jetzt C# lernen sollen, wenn doch Visual Basic und auch C++ an das .net-Framework angepasst werden, also die gleichen Features nutzen können. Das ist zwar richtig, allerdings sind C++ bzw. Visual Basic angepasste Sprachen, während es sich bei C# um die Systemsprache des .net-Frameworks handelt. D.h. keine andere Sprache unterstützt das.net-Framework so, wie es C# tut. Und keine andere Sprache basiert in dem Maße auf .net wie C#. Wenn es um die Programmierung neuer Komponenten für das .net-Framework geht, ist C# daher die erste Wahl.

### 1.3.4 IL-Code und JIT-Compilierung

Da .net sowohl unabhängig vom Betriebssystem als auch von der Programmiersprache sein sollte, musste auch ein anderer Weg der Compilierung gefunden werden. Wenn Sie mit einer der etablierten Sprachen gearbeitet und eines Ihrer Programme compiliert haben, dann wurde dieses Programm für genau das Betriebssystem compiliert, auf dem es auch erstellt wurde, denn es wurden die Funktionen genau dieses Betriebssystems benutzt. Das können Sie sich ungefähr so vorstellen, als ob dieses Buch ins Englische übersetzt werden würde – ein Franzose, der die englische Sprache nicht beherrscht, könnte dann natürlich nichts damit anfangen.

Aus diesem Grund wird von den neuen, auf .net basierenden Compilern ein so genannter Zwischencode erzeugt, der *Intermediate Language Code* oder kurz *IL-Code*. Dieser hat nun den Vorteil, dass das .net-Framework ihn versteht und in eine Sprache übersetzen kann, die das darunter liegende Betriebssystem versteht. Um bei dem Beispiel mit dem Franzosen und dem englischen Buch zu bleiben, könnte man sagen, das Framework wirkt wie ein Simultan-Dolmetscher.

Auf diese Art und Weise zu compilieren ist auf verschiedenen Wegen möglich. Man könnte bereits bei der Installation eines Programms compilieren, so dass das Programm direkt in fertiger Form vorliegt und gestartet werden kann. Man könnte ebenso die einzelnen Programmteile genau dann compilieren, wenn sie gebraucht werden. Diese Vorgehensweise nennt man auch JIT-Compilierung (Just-In-Time, Compilierung bei Bedarf).

Das .net-Framework benutzt beide Möglichkeiten. Die Komponenten des Frameworks werden bei der Installation compiliert und liegen dann als ausführbare Dateien vor, während die Programme *Just-In-Time* compiliert werden. Diese Art von Compiler nennt man in der Kurzform *Jitter*, und das .net-Framework bringt zwei davon mit. Leider ist in der neuesten Version von .net kein spezielles Konfigurationsprogramm für die Jitter mehr enthalten, in der ersten Version, den Next Generation Windows Services, war das noch der Fall.

## 1.4 Editoren für C#

Zum Programmieren benötigt man auch einen Editor, denn irgendwo muss man ja den Programmtext eingeben. Leider liegt dem .net-Framework kein Editor bei, nicht mal eine Basis-Version eines Editors. Die optimale Lösung wäre natürlich das Visual Studio .net,

allerdings wird dies nach vorsichtigen Schätzungen erst gegen Ende des Jahres erwartet.

Glücklicherweise gibt es aber auch noch das Internet, und dort findet man bereits einige Editoren für C#. Grundsätzlich tut es aber jedes Programm, das puren Text verarbeiten kann, also auch der Windows-Editor.

### 1.4.1 Der Windows-Editor

Mit diesem Editor ist es bereits möglich, C#-Code zu schreiben und zu speichern. Allerdings hat diese Lösung, wenn es auch die billigste ist, einige gravierende Nachteile. Zunächst beherrscht der Windows-Editor keine Syntaxhervorhebung, die sehr sinnvoll wäre. Zum Zweiten kann der Compiler für C# nicht eingebunden werden, Sie sind also darauf angewiesen, die Dateien über die Kommandozeile zu compilieren. Alles in allem zwar wie angesprochen die preiswerteste, aber auch schlechteste Lösung.

### 1.4.2 CSharpEd von Antechinus

*CSharpEd* ist ein Editor, der Syntaxhervorhebung und die Einbindung des Compilers unterstützt. Das Programm ist allerdings Shareware, und für die Registrierung schlagen immerhin 30$ zu Buche. Die Shareware-Version dürfen Sie dafür aber zehn Mal nutzen, bevor Sie sich registrieren müssen.

Wenn Ihr Computer ständig läuft (so wie meiner), können Sie die Nutzungszeit auch verlängern, indem Sie das Programm einfach nicht beenden. Die Zeit, die das Programm am Laufen ist, wird nicht kontrolliert, lediglich die Starts werden gezählt. Dieser Tipp kommt übrigens nicht von mir, sondern vom Autor des Programms.

Abbildung 1.2 zeigt ein Bild von *CSharpEd* zur Laufzeit. Im Internet können Sie die aktuellste Version der Software unter der Adresse *www.c-point.com* herunterladen. Die Sharewareversion, so aktuell, wie es mir möglich war, finden Sie auch auf der Buch-CD.

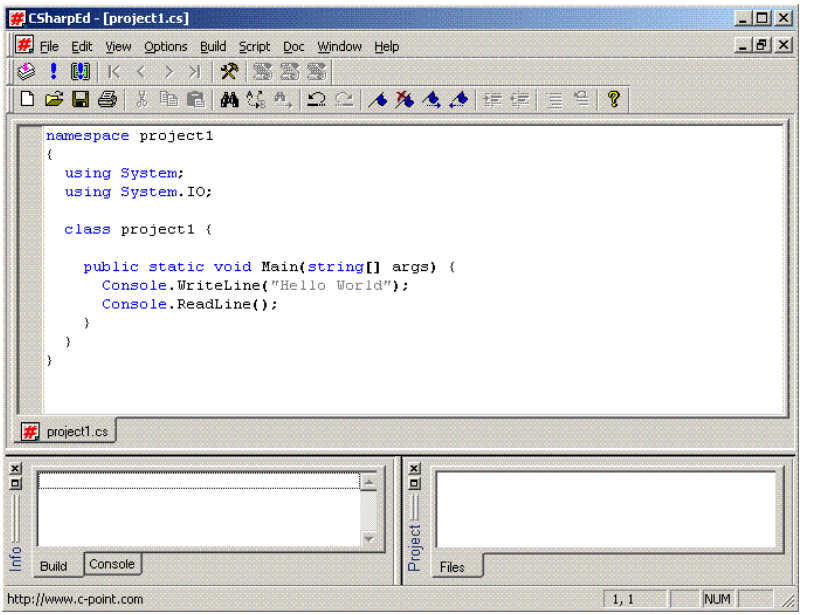


*Abbildung 1.1: CSharpEd in Aktion*

### 1.4.3 SharpDevelop von Mike Krüger

Nicht lachen, er heißt wirklich so. Hat aber nichts mit dem Typ zu tun, den Sie aus dem Fernsehen kennen.

*SharpDevelop* ist Freeware unter GNU-Lizenz, d. h. Sie erhalten zusammen mit dem Programm den kompletten Quellcode und dürfen diesen auch verändern, wenn Sie die Änderungen dann auch öffentlich zur Verfügung stellen. Das Besondere an diesem Editor ist, dass er komplett in C# geschrieben ist. Die aktuelle Version zu der Zeit, zu der ich dieses Buch schreibe, ist die Version 0.5. Sie finden auch diesen Editor in der neuesten Version auf der Buch-CD.

SharpDevelop ermöglicht die Compilierung des Quellcode auf Tastendruck (über die Taste F5) und besitzt auch sonst einige interessante Features. Einer der wichtigsten Vorteile ist, dass der Editor frei erhältlich ist und dass ständig daran gearbeitet wird. Natürlich finden Sie auch diesen Editor auf der Buch-CD. Im Internet finden Sie die jeweils aktuellste Version unter der Adresse *http:/www.icsharpcode.net/* .

Abbildung 1.2 zeigt ein Bild von *SharpDevelop* zur Laufzeit.

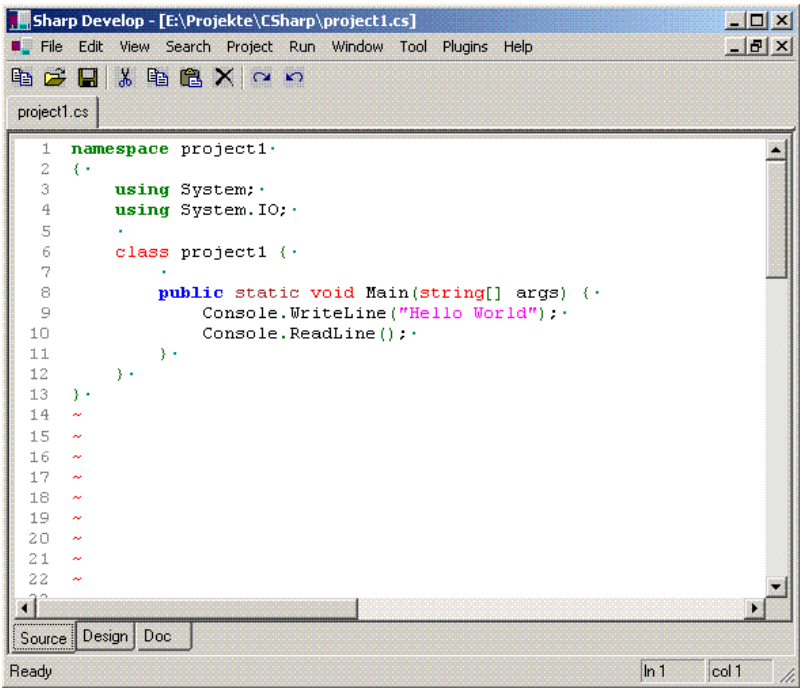


*Abbildung 1.2: SharpDevelop im Einsatz*

### 1.4.4 Visual Studio 6

Sie haben auch die Möglichkeit, falls Sie im Besitz von Visual Studio 6 sind, dieses zur Erstellung von C#-Programmen zu nutzen. Der Nachteil, der sich dabei ergibt, ist das Fehlen von Syntaxhervorhebung für C#-Dateien (da das Programm diese nicht kennt) und die eher schlechte Einbindung des Compilers, der nur jeweils die aktuell zum Bearbeiten geöffnete Datei compiliert. Dennoch ist auch dies eine Alternative. Und nach einigem Tüfteln findet man auch heraus, wie man die Oberfläche von VS6 dazu überreden kann, die Syntax von C#-Programmen zu erkennen und hervorzuheben.

#### Syntaxhervorhebung anpassen

Für die Syntaxhervorhebung müssen wir ein wenig tricksen, und damit meine ich wirklich tricksen. Insbesondere müssen wir jetzt auf die Registry zugreifen, in der die Dateitypen registriert sind, für die die Syntaxhervorhebung gelten soll. Wir werden der Entwicklungsumgebung für C++ beibringen, auch die Syntax von C# zu erkennen.

Das ist deshalb sinnvoll, da C++ bereits viele Schlüsselwörter enthält, die auch in C# Verwendung finden.

Starten Sie das Programm REGEDIT.EXE über START/AUSFÜHREN. Suchen Sie jetzt den Schlüssel

HKEY_CURRENT_USER\SOFTWARE\MICROSOFT\DEVSTUDIO\6.0\
TEXT EDITOR\TABS/LANGUAGE SETTINGS\C/C++\FILEEXTENSIONS

Zugegebenermaßen ein ziemlich langes Elend von einem Registry-Schlüssel. Sie finden dort eine Zeichenkette, die alle Endungen der Programme enthält, für die Syntaxhervorhebung angewandt werden soll. C#-Dateien haben die Dateiendung „.cs“, die in der Liste naturgemäß fehlt. Fügen Sie sie einfach hinzu. Fortan wird die Entwicklungsumgebung auch C#-Dateien als solche erkennen, für die eine Syntaxhervorhebung gilt. Nun benötigen wir nur noch die Schlüsselwörter von C#.

Die hervorzuhebenden Schlüsselwörter finden sich in der Datei USERTYPE.DAT, die sich im gleichen Verzeichnis befindet wie die ausführbare Datei der Entwicklungsumgebung (MSDEV.EXE). Diese müssen wir nun ändern, falls sie existiert, bzw. erstellen, wenn sie noch nicht existiert. Sie finden eine für C# angepasste Datei auf der Buch-CD im Verzeichnis EDITOREN\VS6\. Kopieren Sie diese Datei nun in das gleiche Verzeichnis, in dem sich auch MSDEV.EXE befindet. Eine evtl. vorhandene USERTYPE.DAT-Datei wird dabei überschrieben, es empfiehlt sich daher, sie vorher zu sichern.

Wenn Sie jetzt mit C#-Dateien arbeiten, werden Sie feststellen, dass die Entwicklungsumgebung diese Dateien erkennt und die Syntax entsprechend den Schlüsselwörtern von C# hervorhebt.

Auf der CD finden Sie außer der benötigten Datei USERTYPE.DAT mit den Erweiterungen der Schlüsselwörter auch eine Datei CS.REG, die den Registry-Eintrag für Visual Studio 6 enthält. Wenn Sie auf diese Datei doppelklicken, wird der benötigte Registry-Eintrag geändert, ohne dass Sie von Hand eingreifen müssen.

### Einbinden des Compilers

Jetzt müssen wir noch den Compiler einbinden. Das Menü Extras der Entwicklungsumgebung kann angepasst werden, diese Möglichkeit nutzen wir jetzt aus, um den Compiler hinzuzufügen.

– Wählen Sie den Menüpunkt EXTRAS, dann den Unterpunkt ANPASSEN. Es erscheint ein Dialog, in dem Sie auf die Seite EXTRAS wechseln müssen. Dort finden Sie eine Liste der Menüeinträge. Rollen

Sie die Liste bis ganz nach unten und klicken Sie auf die letzte Zeile. Sie können nun die Beschriftung des neuen Menüpunkts eingeben, z.B. „C# compilieren“.

- Im Eingabefeld BEFEHL geben Sie bitte *cmd.exe* ein.
- Im Eingabefeld ARGUMENTE geben Sie bitte */c csc "$(FilePath)" && "$(FileName)"* ein.
- Markieren Sie bitte auch das Feld FENSTER "AUSGABE" VERWENDEN.

Sie sind jetzt vorbereitet, um C#-Programme zu compilieren. Sie finden den neuen Menüpunkt im EXTRAS-Menü, allerdings wird wie angesprochen nur die jeweils aktive Datei kompiliert.

Das kompilierte Programm wird in einem Ausgabefenster ausgeführt, wenn der Compiler erfolgreich war. Ansonsten sehen Sie dort die Fehlermeldungen, die der Compiler zurückliefert. Diese haben das gleiche Format wie die Fehlermeldungen, die auch VS6 liefert, Sie sollten damit also keine Probleme haben.

### 1.4.5 Visual Studio .net Beta 1

Falls Sie sich die Beta-Version des neuen Visual Studio bereits zugelegt haben, haben Sie damit natürlich eine optimale Umgebung, um mit C# zu programmieren. Wenn Sie mit dieser Software ein neues Projekt starten, werden in der Regel die erste Klasse des Projekts, ein eigener Namensraum und die Methode `Main()` als Einsprungpunkt bereits deklariert. Außerdem ist natürlich die Anbindung an den Compiler optimal und die Möglichkeiten des integrierten Debuggers sind auch nicht zu unterschätzen. Weitere Vorteile sind die Ausgabe von Fehlermeldungen und die Tatsache, dass Sie die `WinForms`-Bibliothek mit den visuellen Steuerelementen direkt benutzen können.

Visual Studio .net ist allerdings für die Lektüre dieses Buches nicht zwingend erforderlich. In diesem Buch wird mehr auf die Sprache eingegangen, und prinzipiell ist es möglich, jedes vorgestellte Programm auch mit dem Windows-Editor zu erstellen. Ein Buch über die Programmierung mit dem Visual Studio .net ist bereits geplant und wird verfügbar sein, wenn die Entwicklungssoftware das Veröffentlichungsstadium erreicht hat.

Zur Eingabe der Quelltexte aus dem Buch sollten Sie stets eine neue Konsolenanwendung starten. Wählen Sie dazu entweder von der Startseite des VisualStudio CREATE NEW PROJECT oder aus dem Menü FILE den Menüpunkt NEW | PROJECT (Tastenkürzel Strg+⇧+N). Aus dem erscheinenden Dialog wählen Sie dann VISUAL C# PROJECTS und rechts daneben als Template CONSOLE APPLICATION. Es kann ei-

nige Zeit dauern (zumindest beim ersten Projekt nach dem Start der Entwicklungsoberfläche), bis Sie mit der Programmierung beginnen können, dafür haben Sie aber eine optimale Hilfefunktion und den Vorteil, die vom Compiler während des Übersetzungsvorgangs bemängelten Fehler direkt anspringen und korrigieren zu können. Alles in allem lohnt sich die Bestellung des Softwarepakets, zumal der Veröffentlichungstermin noch nicht feststeht. Und wenn man Microsofts Wartezeiten kennt, weiß man, dass es sich eher um das dritte oder vierte Quartal 2001 handeln wird.

## 1.5 Die CD zum Buch

Auf der CD finden Sie ein große Anzahl von Beispielen, die auch im Buch behandelt werden.Außerdem habe ich noch einige Editoren aus dem Internet daraufgepackt. Das .net-Framework wollte ich eigentlich auch noch mitliefern, das war aber aufgrund einer fehlenden Erlaubnis von Microsoft nicht möglich.

Alle Programme wurden mit einem herkömmlichen Texteditor verfasst, nicht mit der Visual Studio-Software. Der Grund hierfür war, dass das Visual Studio so genannte Solution-Dateien anlegt und auch eine eigene Verzeichnisstruktur für die einzelnen Programme erstellt. Da das für die hier verfassten kleinen Applikationen nicht notwendig ist, habe ich mich für einen herkömmlichen Texteditor entschieden. Alle Programme sollten auch ohne Visual Studio (aber mit installiertem .net-Framework) laufen.

Damit wären wir am Ende der Einführung angekommen. Bisher war alles zugegebenermaßen ein wenig trocken, dennoch haben Sie schon jetzt eine gewisse Übersicht über die Laufzeitumgebung und die Art und Weise des Vorgehens beim Compilieren von Programmen kennen gelernt. Außerdem erhielten Sie einen Überblick über die verfügbaren Editoren von C# und können, wenn Sie es wollen, das Visual Studio 6 so anpassen, dass es C#-Dateien versteht und ihre Syntax hervorheben kann.

Wir werden nun langsam in die Programmierung einsteigen. Für diejenigen, die bereits in einer anderen Programmiersprache programmiert haben, mag manches sehr vertraut erscheinen. Vor allem Java-Programmierer werden sehr viele bekannte Dinge wiederentdecken. Dennoch unterscheidet sich C# sowohl von C++ als auch von Java, allerdings mehr im „Inneren“. Umsteiger von anderen Sprachen sollten sich weiterhin vor Augen halten, dass dieses Buch für den kompletten Neueinsteiger konzipiert wurde, und es dem Autor daher nachsehen, wenn die eine oder andere Sache gar zu umfangreich erklärt wird.

# 2 Erste Schritte

## 2.1 Grundlagen

Bevor wir zum ersten Programm kommen, möchte ich zunächst einige Worte über Programmierung allgemein verlieren. Diese sind natürlich vor allem für den absoluten Neueinsteiger interessant. Diejenigen unter den geschätzten Lesern, die bereits mit einer anderen Programmiersprache gearbeitet haben, können diesen Abschnitt überspringen.

### 2.1.1 Algorithmen und Programme

Das Programmieren besteht in der Regel darin, einen Algorithmus in ein Programm umzusetzen. Ein Algorithmus beschreibt dabei ein Verfahren, mit dem ein Problem gelöst werden kann. Dabei ist dieser Algorithmus vollkommen unabhängig vom verwendeten Computer oder der verwendeten Programmiersprache, es handelt sich vielmehr um eine allgemeingültige Verfahrensweise, die zur Lösung eines Problems führt.

Die Aufgabe des Programmierers ist nun die Umsetzung dieses Algorithmus in eine Folge von Anweisungen, die der Computer versteht und mit deren Hilfe er das durch den Algorithmus vorgegebene Ergebnis erzielt. Natürlich kann ein Algorithmus in verschiedenen Programmiersprachen umgesetzt werden, normalerweise sucht man sich die für das betreffende Problem günstigste heraus.

***Systemabhängigkeit***

In dem Moment, wo ein Algorithmus in eine Programmiersprache umgesetzt wird, wird das entstehende Programm normalerweise systemabhängig. So kann ein Programm, das unter Windows erstellt wurde, in der Regel auch nur auf einem anderen Computer ausgeführt werden, der ebenfalls Windows als Betriebssystem enthält. Unter Linux laufen solche Programme beispielsweise nicht. Eine solche

Betriebssystemabhängigkeit gilt eigentlich für alle derzeit bekannten Programmiersprachen. Mit ein Grund dafür ist, dass in der Regel bereits fertige Routinen, die vom Betriebssystem selbst zur Verfügung gestellt werden, bei der Programmierung einer Applikation Verwendung finden.

***Systemunabhängigkeit***

Mit C# bzw. dem .net-Framework soll sich das grundlegend ändern. Programme, die basierend auf .net entwickelt werden (mit welcher Programmiersprache auch immer) sind überall dort lauffähig, wo das .net-Framework installiert ist. Um dies zu erreichen, wird von den jeweiligen Compilern ein Zwischencode erzeugt, der weder Quellcode noch fertig compilierter Code ist, aber vom .net-Framework in die fertige Version übersetzt wird, unabhängig vom Betriebssystem. Das .net-Framework wird sozusagen zwischen Programm und Betriebssystem geschoben. Bekannt ist diese Vorgehensweise ja bereits von Java.

***Programmieren***

Bei der Erstellung eines Programms sind für den Programmierer einige Dinge wichtig, auch wenn sie sich teilweise erst sehr viel später auswirken. Zunächst sollte man sich einen sauberen Programmierstil angewöhnen. Wichtig ist vor allem, dass Programme auch nach längerer Zeit noch einwandfrei wartbar sind und dass unter Umständen andere Programmierer auch mit dem Quellcode zurechtkommen. Ich werde im nächsten Abschnitt näher darauf eingehen.

Ein weiterer Punkt ist, dass es sehr viele Routinen gibt, die bereits all das tun, was man in einer Methode als Funktionalität bereitstellen will. Man sollte nie das Rad neu erfinden wollen – falls also eine Methode existiert, die das tut, was Sie ohnehin programmieren wollen, benutzen Sie sie ruhig. Der Stichpunkt ist die Wiederverwendbarkeit, auch hierauf werde ich noch ein wenig eingehen.

### 2.1.2 Programmierstil

Ein guter Programmierstil ist sehr wichtig, um ein Programm letzten Endes erfolgreich zu machen. Ein guter Programmierstil bedeutet, ein Programm so zu erstellen, dass es

- gut lesbar und damit auch gut zu warten ist,
- so wenig Fehler wie möglich enthält,
- Fehler des Anwenders abfängt, ohne dass irgendein Schaden entsteht (bzw. sogar ohne dass der Anwender bemerkt, dass er einen Fehler gemacht haben könnte),
- so effektiv und schnell wie irgend möglich arbeitet.

All diese Kriterien werden bei weitem nicht immer erfüllt, man sollte aber versuchen, sie so gut wie möglich in eigenen Programmen umzusetzen. Schon das erste Kriterium – nämlich eine Programmierung, die gut zu lesen ist – ist eine sehr wichtige Sache, auch wenn manche Programmierer es nicht einsehen.

Lesbarkeit

Es gibt nichts Schlimmeres als ein Programm, bei dem man sich durch den Quelltext kämpfen muss wie durch einen Urwald. Wichtig ist hierbei nicht immer nur, dass man selbst den Programmtext gut lesen kann, sondern dass auch andere Programmierer, die möglicherweise irgendwann einmal mit dem Quelltext in Berührung kommen und eine Änderung programmieren müssen, problemlos damit arbeiten können. Ein guter Programmierstil wäre z.B. folgender:

```
namespace Test
{
  public class MainClass
  {
    public static void Main()
    {
      int x;
      int y;
      int z;
      x = Console.ReadLine().ToInt32();
      y = Console.ReadLine().ToInt32();
      z = x/y;
      Console.WriteLine("Ergebnis: {0}",z);
    }
  }
}
```

In obigem Beispiel wurde für jede Anweisung eine einzelne Zeile vorgesehen (was in C# nicht notwendig ist), die einzelnen Programmblöcke wurden sauber voneinander getrennt (man erkennt sie an den geschweiften Klammern) und die Zusammengehörigkeiten sind klar sichtbar. Das Programm lässt den Anwender zwei Zahlen eingeben und dividiert diese, wobei es sich in diesem Fall um ganze Zahlen handelt, das Ergebnis also ebenfalls keine Nachkommastellen besitzt. Sehen Sie sich nun folgenden Code an:

```
namespace Test {
public class MainClass {
public static void Main() {
int x;int y;int z;x=Console.ReadLine().ToInt32();
y=Console.ReadLine().ToInt32();z=x/y;
Console.WriteLine("Ergebnis: {0}",z);}}}
```

Die Funktion beider Programme ist exakt die gleiche. Sie werden auch beide problemlos vom Compiler akzeptiert. Aber seien Sie mal ehrlich: Welches der Programme können Sie besser lesen?

Bei der Erstellung von Programmen sollten Sie stets darauf achten, wirklich für jeden Befehl eine eigene Zeile zu benutzen, die einzelnen Programmblöcke entsprechend ihrer Zusammengehörigkeit einzurücken und auch Leerzeilen zu benutzen, um die Trennung der einzelnen Programmblöcke etwas hervorzuheben. Auf das Laufzeitverhalten der fertigen Applikation haben diese gestalterischen Elemente später keinen Einfluss.

### 2.1.3 Fehlerbeseitigung

Wenn ein Anwender ein Programm erwirbt, kann er in der Regel davon ausgehen, dass dieses Programm so wenig Fehler wie möglich enthält. Bei den großen Firmen gibt es dafür die so genannten *Beta*-Versionen, die an ausgewählte Personen verteilt und von diesen getestet werden, um auch die letzten gravierenden Fehler noch zu finden. Ebenso bekannt sind die umfangreichen Patches, die oftmals im Internet erhältlich sind, oder auch Service-Packs, die sowohl Fehlerbereinigung betreiben, als auch neue Features zu den Programmen hinzufügen.

Allerdings werden Sie wohl keine derart umfangreichen Applikationen entwerfen (zumindest nicht am Anfang) und auch Ihren Beta-Test müssen Sie möglicherweise selbst durchführen. Sie sollten damit allerdings nicht warten, bis das Programm fertig ist, sondern schon während der Programmentwicklung mit den Testläufen beginnen.

***Testläufe***

Testen Sie ein Programm so oft wie möglich und achten Sie darauf, auch die scheinbar unwichtigen Teile stets zu überprüfen. Die meisten Fehler schleichen sich dort ein, wo man sich zu sicher ist, weil man eine Methode schon sehr oft programmiert hat. Schon ein kleiner Schreibfehler kann zu einem unerwünschten Ergebnis führen, auch wenn das Programm ansonsten korrekt ausgeführt wird. Führen Sie auch die Funktionen aus, die normalerweise recht selten benutzt werden (denn diese werden bei einem Test oftmals vergessen). Es wird immer mal wieder einen Benutzer geben, der gerade diese Funktion benötigt, und wenn Sie dann an dieser Stelle einen Fehler im Programm haben, ist das zu Recht ärgerlich.

***Fehler abfangen***

Der Benutzer einer Applikation ist ohnehin die große Unbekannte in der Rechnung. Sie sollten immer damit rechnen, dass es sich um eine Person handelt, die sich mit dem Computer oder mit dem Betriebs-

system nicht so gut auskennt und daher auch Dinge macht, die Sie selbst (weil Sie sich auskennen) nie tun würden. Das Beispiel aus Abschnitt 2.1.2 wäre schon allein deshalb nicht sicher, weil es möglich wäre, dass der Anwender an Stelle einer Zahl eine andere Zeichenfolge eingibt. Um das Programm wirklich sicher zu machen, müssen Sie dies entweder verhindern (so dass der Anwender überhaupt nicht die Möglichkeit hat, etwas anderes als Zahlen einzugeben) oder Sie müssen eine fehlerhafte Eingabe abfangen und dem Anwender nach Ausgabe einer entsprechenden Fehlermeldung die erneute Eingabe ermöglichen. Fehler, in C# *Exceptions* genannt, können Sie mit so genannten Schutzblöcken abfangen, auf die wir in Kapitel 11 noch zu sprechen kommen.

### 2.1.4 Wiederverwendbarkeit

Wiederverwendbarkeit ist ein gutes Stichwort, weil es beim Programmieren in der heutigen Zeit unabdingbar ist. Natürlich werden Sie nicht alle Funktionen immer von Grund auf neu programmieren, sondern stattdessen auf bereits vorhandene Funktionen zurückgreifen. Nichts anderes tun Sie, wenn Sie eine der vielen Klassen des .net-Frameworks benutzen.

Anders herum ist es aber auch so, dass Sie eine eigene Klasse oder einen eigenen Programmteil auch wiederverwenden können, z.B. wenn es sich um eine Klasse zur Speicherung von Optionen handelt. So wäre es durchaus möglich (und sinnvoll), eine eigene Klasse zu erstellen, die das Speichern der Programmoptionen für Sie erledigt, wobei Sie selbst beim ersten Aufruf den Namen des Programms und den Speicherort der Optionen in der Registry selbst angeben können.

Wiederverwendbarkeit ist also ein sehr wichtiger Faktor bei der Programmierung. Wichtig ist wie gesagt, das Rad nicht neu zu erfinden – wenn es eine Funktion gibt, die Ihre Bedürfnisse erfüllt, benutzen Sie sie ruhig. Sie sind nicht gezwungen, die gleiche Funktionalität nochmals zu programmieren, wenn sie in einer DLL oder in einem anderen Programmteil bereits zur Verfügung gestellt wird. Auf der anderen Seite haben Sie natürlich die Möglichkeit, eigene Klassen in Form einer DLL anderen Programmierern zur Verfügung zu stellen.

*Ein erstes Programm*

Nach diesen Grundlagen wollen wir nun ein erstes Programm in C# erstellen. Natürlich haben die hier vorgestellten grundlegenden Vorgehensweisen auch in den diversen Beispielen des Buchs Verwendung gefunden, allerdings wird das oftmals nicht so deutlich, da es sich doch eher um kleine Programme handelt. Erst bei umfangreichen Applikationen werden Sie es zu schätzen wissen, wenn Sie bei

der Erstellung eines Programms sorgsam planen oder, falls notwendig, auch mal einen Programmteil komplett auseinander reißen, weil Sie kleinere „Häppchen“ erstellen wollen, die dafür aber wiederverwendbar sind und für andere Applikationen eine sinnvolle Erweiterung darstellen.

## 2.2 Hallo Welt die Erste

Das erste Programm, das ich Ihnen an dieser Stelle vorstellen möchte, ist das wohl populärste Programm, das jemals auf einem Computer erstellt wurde. Populär deshalb, weil es vermutlich in jeder Programmiersprache implementiert wurde und auch unter jedem Betriebssystem. Ganz recht, es handelt sich um das allseits bekannte *Hallo-Welt*-Programm.

### 2.2.1 Der Quelltext

Das Programm gibt die Nachricht *Hallo Welt* in einer DOS-Box aus und wartet dann, bis der Anwender die Taste ↵ drückt.

```
namespace HalloWelt
{

/* Hallo Welt Konsolenapplikation */
/* Autor:   Frank Eller           */
/* Sprache: C#                    */

  using System;

  public class HalloWelt1
  {
    public static int Main(string[] args)
    {
      Console.WriteLine("Hallo Welt");
      return 0;
    }
  }
}
```

Geben Sie das Programm in den von Ihnen bevorzugten Editor ein und speichern Sie es unter dem Namen HELLOWORLD.CS. Falls Sie mit *SharpDevelop* arbeiten, genügt nach der Eingabe der Zeilen ein Druck auf die Taste F5 um das Programm zu kompilieren. Falls Sie bereits mit Visual Studio .net (Beta 1) arbeiten, benutzen Sie entweder die Tastenkombination Strg+F5 oder starten das Programm über den

Menüpunkt DEBUG/START WITHOUT DEBUGGING. Bei anderen Editoren (vor allem beim Windows Editor) müssen Sie die Compilierung von Hand zu Fuß starten.

***Compilieren des Programms***

Das Programm, das die Compilierung durchführt, heißt CSC.EXE. Üblicherweise finden Sie es im Versionsverzeichnis des installierten .net-Framework, das ist aber nicht so wichtig. Wenn .net richtig installiert ist (und das ist der Regelfall) dann findet das System den Compiler auch von sich aus anhand der Dateiendung.

Starten Sie die Eingabeaufforderung (im Startmenü unter ZUBEHÖR). Wechseln Sie nun in das Verzeichnis, in das Sie die Datei HELLOWORLD.CS gespeichert haben, und starten Sie die Compilierung durch die Eingabe von

```
CSC HELLOWORLD.CS
```

Normalerweise sollte der Compiler ohne Murren durchlaufen. Die ausgegebene Datei trägt den Namen HELLOWORLD.EXE. Wenn Sie dieses Programm starten, wird der Schriftzug

```
Hallo Welt
```

auf Ihrem Bildschirm ausgegeben. Sie haben damit Ihr erstes C#-Programm erfolgreich erstellt.

Der Kommandozeilencompiler CSC.EXE ermöglicht die Eingabe mehrerer voneinander unabhängiger Optionen als Kommandozeilenparameter. Eine Übersicht über die Möglichkeiten finden Sie im Anhang. Für den Moment jedoch genügt die einfache Compilierung, wie wir sie hier durchgeführt haben.

Sie finden das Programm auch auf der CD im Verzeichnis BEISPIELE\KAPITEL_2\HALLOWELT1\.

### 2.2.2 Blöcke

***{ und }***

Im Beispiel ist deutlich eine Untergliederung zu sehen, die in C# mit den geschweiften Klammern durchgeführt wird. Alle Anweisungen innerhalb geschweifter Klammern werden im Zusammenhang als eine Anweisung angesehen. Geschweifte Klammern bezeichnen in C# also einen Programmblock.

Mit Hilfe dieser Programmblöcke werden die einzelnen zusammengehörenden Teile eines Programms, wie Namensräume, Klassen, Schleifen, Bedingungen, Methoden usw. voneinander getrennt. Programmblöcke dienen also auch als eine Art „roter Faden“ für den Compiler.

Mit Hilfe dieses roten Fadens wird dem Compiler angezeigt, wo ein Programmteil beginnt, wo er endet und welche Teile er enthält. Da Programmblöcke mehrere Anweisungen sozusagen zusammenfassen und sie wie eine aussehen lassen, ist es auch möglich, an Stellen, an denen eine Anweisung erwartet wird, einen Anweisungsblock zu deklarieren. Für den Compiler sieht der Block wie eine einzelne Anweisung aus, obwohl es sich eigentlich um die Zusammenfassung mehrerer Anweisungen handelt.

Innerhalb einer Methode können Sie Anweisungsblöcke auch dazu benutzen, die einzelnen Teile optisch voneinander zu trennen. Für den Compiler macht das keinen Unterschied, das Programm wird nicht schneller und nicht langsamer. Wenn Sie z.B. am Anfang der Methode eine Art „Initialisierungsteil“ haben und dann den eigentlichen funktionellen Teil, könnten Sie das auch so programmieren:

```
public void Ausgabe()
{
  //Variablen initialisieren
  {
    //Anweisungen
  }

  //Funktionen
  {
    //Anweisungen
  }
}
```

Die beiden doppelten Schrägstriche im obigen Quelltext (der ja eigentlich gar keiner ist) bezeichnen einen Kommentar und dienen der Verdeutlichung. Im Prinzip können Sie Programmblöcke überall anwenden, wo Sie gerade wollen. Aber wo wir gerade bei den Kommentaren sind ...

### 2.2.3 Kommentare

Das *Hallo-Welt*-Programm enthält drei Zeilen, die nicht compiliert werden und nur zur Information für denjenigen dienen, der den Quelltext bearbeitet. Es handelt sich dabei um die drei Zeilen

```
/* Hallo Welt Konsolenapplikation */
/* Autor:   Frank Eller           */
/* Sprache: C#                    */
```

Es handelt sich um einen Kommentar, einen Hinweis, den der Programmierer selbst in den Quelltext einfügen kann, der aber nur zur

Information dient, das Laufzeitverhalten des Programms nicht verändert und auch ansonsten keine Nachteile mit sich bringt.

**/* und */**

Die Zeichen /* und */ stehen in diesen Zeilen für den Anfang bzw. das Ende des Kommentars. Es ist aber in diesem Fall nicht notwendig, so wie ich es hier getan habe, diese Zeichen für jede Zeile zu wiederholen. Das geschah lediglich aus Gründen des besseren Erscheinungsbildes. Man hätte den Kommentar auch folgendermaßen schreiben können:

```
/*
  Hallo Welt Konsolenapplikation
  Autor:   Frank Eller
  Sprache: C#
*/
```

Alles, was zwischen diesen Zeichen steht, wird vom Compiler als Kommentar angesehen. Allerdings können diese Kommentare nicht verschachtelt werden, denn sobald der innere Kommentar zuende wäre, wäre gleichzeitig der äußere auch zuende. Die folgende Konstellation wäre also nicht möglich:

```
/*
    Hallo Welt Konsolenapplikation
/*  Autor:   Frank Eller          */
    Sprache: C#
*/
```

Das Resultat wäre eine Fehlermeldung des Compilers, der die Zeile

```
    Sprache: C#
```

nicht verstehen würde.

**//**

Es existiert eine weitere Möglichkeit, Kommentare in den Quelltext einzufügen. Ebenso wie in C++, Java oder Delphi können Sie den doppelten Schrägstrich dazu verwenden, einen Kommentar bis zum Zeilenende einzuleiten. Alle Zeichen nach dem doppelten Schrägstrich werden als Kommentar angesehen, das Ende des Kommentars ist das Zeilenende. Es wäre also auch folgende Form des Kommentars möglich gewesen:

```
// Hallo Welt Konsolenapplikation
// Autor:   Frank Eller
// Sprache: C#
```

**Kommentare verschachteln**

Die verschiedenen Kommentare – einmal den herkömmlichen über mehrere Zeilen gehenden und den bis zum Zeilenende – können Sie durchaus verschachteln. Es wäre also auch möglich gewesen, den Kommentar folgendermaßen zu schreiben:

```
/*
   Hallo Welt Konsolenapplikation
// Autor:   Frank Eller
   Sprache: C#
*/
```

Sinn und Zweck dieser Kommentare ist es, dem Programmierer eine bessere Übersicht über die einzelnen Funktionen eines Programms zu geben. Oftmals ist es so, dass eine Methode recht kompliziert ist und man im Nachhinein nicht mehr so recht weiß, wozu sie eigentlich dient. Dann sind Kommentare ein hilfreiches Mittel, um auch nach längerer Zeit einen Hinweis auf die Funktion zu geben.

### 2.2.4 Die Methode Main()

Anhand des Quelltextes ist bereits ersichtlich, dass die geschweiften Klammern einen Programmblock darstellen. Die Klasse `HalloWelt1` gehört zum Namespace `HalloWelt`, die Methode `Main()` wiederum ist ein Bestandteil der Klasse `HalloWelt1`. Mit dieser Methode wollen wir auch beginnen, denn sie stellt die Hauptmethode eines jeden C#-Programms dar.

Ein C#-Programm kann (normalerweise) nur eine Methode `Main()` besitzen. Diese Methode muss sowohl als öffentliche Methode deklariert werden als auch als statische Methode, d.h. als Methode, die Bestandteil der Klasse selbst ist. Wir werden im weiteren Verlauf des Buches noch genauer darauf eingehen.

***public und static***

Die beiden reservierten Wörter **public** und **static** erledigen die notwendige Definition für uns. Dabei handelt es sich um so genannte *Modifikatoren*, die wir im weiteren Verlauf des Buches noch genauer behandeln werden. **public** bedeutet, dass die nachfolgende Methode öffentlich ist, d.h. dass von außerhalb der Klasse, in der sie deklariert ist, darauf zugegriffen werden kann. **static** bedeutet, dass die Methode Bestandteil der Klasse selbst ist. Damit muss, um die Methode aufzurufen, keine Instanz der Klasse erzeugt werden.

Was es mit der Instanziierung bzw. Erzeugung eines Objekts im Einzelnen auf sich hat, werden wir in Kapitel 3 noch genauer besprechen. An dieser Stelle genügt es, wenn Sie sich merken, dass man die Methode einfach so aufrufen kann, wenn man den Namen der Klasse und der Methode weiß. Für uns ist das die Methode `Main()` betreffend allerdings ohnehin unerheblich, da die Laufzeitumgebung diese automatisch bei Programmstart aufruft.

`Main()` ist deshalb so wichtig und muss deshalb auf diese Art deklariert werden, weil sie den Einsprungpunkt eines Programms darstellt. Für die Laufzeitumgebung bedeutet dies, dass sie, sobald ein C#-Programm gestartet wird, nach eben dieser Methode sucht und die darin enthaltenen Anweisungen ausführt. Wenn das Ende der Methode erreicht ist, ist auch das Programm beendet.

Verständlicherweise ist das auch der Grund, warum in der Regel nur eine Methode mit Namen `Main()` existieren darf – der Compiler bzw. die Laufzeitumgebung wüssten sonst nicht, bei welcher Methode sie beginnen müssten. Wie bereits gesagt, gilt für diese Methode, dass sie mit den Modifikatoren **public** und **static** deklariert werden muss. Außerdem muss der Name groß geschrieben werden (anders als in C++ - dort heißt die entsprechende Methode zwar auch `main`, aber mit kleinem „m“). Der Compiler unterscheidet zwischen Groß- und Kleinschreibung, daher ist die Schreibweise wichtig.

Die Methode `Main()` stellt den Einsprungpunkt eines Programms dar. Aus diesem Grund darf es in einem C#-Programm (normalerweise) nur eine Methode mit diesem Namen geben. Sie muss mit den Modifikatoren **public** und **static** deklariert werden. **public**, damit die Methode von außerhalb der Klasse erreichbar ist, und **static**, damit nicht eine Instanz der Klasse erzeugt werden muss, in der sich die Methode `Main()` befindet.

In welcher Klasse `Main()` programmiert ist, ist wiederum unerheblich.

***mehrere Main()-Methoden***

Wenn hier geschrieben steht, dass es in der Regel nur eine Methode mit Namen `Main()` geben darf, dann existiert natürlich auch eine Ausnahme von dieser Regel. Tatsächlich ist es so, dass Sie wirklich mehrere `Main()`-Methoden deklarieren dürfen, Sie müssen dem Compiler dann aber eindeutig bei der Compilierung mitteilen, welche der Methoden er benutzen soll.

Innerhalb einer Klasse kann es nur eine Methode `Main()` geben. Mit Hilfe eines Kommandozeilenparameters können Sie dem Compiler dann die Klasse angeben, deren `Main()`-Methode als Einsprungpunkt benutzt werden soll. Der Parameter hat die Bezeichnung

```
/main:<Klassenname>
```

Für unseren Fall (wenn es mehrere `Main()`-Methoden im Hallo-Welt-Programm gäbe) würde die Eingabe zur Compilierung also lauten:

```
csc /main:HalloWelt1 HalloWelt.cs
```

Sie geben damit die Klasse an, deren `Main()`-Methode als Einsprungpunkt benutzt werden soll.

Wenn Sie in Ihrer Applikation mehrere `Main()`-Methoden deklariert haben, können Sie dem Compiler mitteilen, welche dieser Methoden er als Einsprungpunkt für das Programm benutzen soll. Das kann für die Fehlersuche recht sinnvoll sein. Normalerweise werden Programme aber lediglich eine Methode `Main()` vorweisen, wodurch der Einsprungpunkt eindeutig festgelegt ist.

***int***

Das reservierte Wort **int**, das direkt auf die Modifikatoren folgt, bezeichnet einen Datentyp, der vor einer Methode für einen Wert steht, den diese Methode zurückliefern kann. Diesen zurückgelieferten Wert bezeichnet man auch als Ergebniswert der Methode.

***return***

Der Ergebniswert wird mit der Anweisung **return** an die aufrufende Methode zurückgeliefert. **return** muss verwendet werden, sobald eine Methode einen Ergebniswert zurückliefern kann, d.h. ein Ergebnistyp angegeben ist. Der Aufruf von **return** liefert nicht nur den Ergebniswert, die Methode wird damit auch beendet.

***void***

`Main()` kann zwei Arten von Werten zurückliefern: entweder einen ganzzahligen Wert des Datentyps **int** oder eben keinen Wert. In diesem Fall wird als Datentyp **void** angegeben, was für eine leere Rückgabe steht. Andere Datentypen sind für `Main()` nicht erlaubt, wohl aber für andere Methoden.

Grundsätzlich lässt sich dazu sagen, dass jede Methode von Haus aus darauf ausgelegt ist, ein Ergebnis (z.B. bei einer Berechnung) zurückzuliefern, aber nicht gezwungen wird, dies zu tun. Wenn die Methode ein Ergebnis zurückliefert, wird der Datentyp des Ergebnisses angegeben. Handelt es sich um eine Methode, die lediglich eine Aktion durchführt und kein Ergebnis zurückliefert, wird der Datentyp **void** benutzt.

Prinzipiell können Sie also jeden Datentyp, auch selbst definierte, als Ergebnistyp verwenden. Dazu werden wir aber im späteren Verlauf des Buchs noch zu sprechen kommen, zunächst wollen wir uns weiter um die grundlegenden Bestandteile eines Programms kümmern.

***Console.WriteLine()***

Kommen wir nun zu den Anweisungen innerhalb der Methode, die ja offensichtlich zur Folge haben, dass ein Schriftzug ausgegeben wird. Da **return** bereits abgehandelt ist, bleibt nur noch die Zeile

```
Console.WriteLine("Hallo Welt");
```

übrig. An dieser Stelle müssen wir ein wenig weiter ausholen. `Console` ist nämlich bereits eine Klasse, eine Konstruktion mit der wir uns noch näher beschäftigen müssen. Klassen bilden die Basis der Programmierung mit C#, alles (auch alle Datentypen) in C# ist eine Klasse.

Die Klasse `Console` steht hier für alles, was mit der Eingabeaufforderung, dem DOS-Fenster zu tun hat. Der Ausdruck *Console* stammt noch aus den Urzeiten der Computertechnik, hat sich aber bis heute gehalten und beschreibt die Eingabeaufforderung nach wie vor treffend. In allen Programmiersprachen wird auch im Falle von Anwendungen, die im DOS-Fenster laufen, von *Konsolenanwendungen* gesprochen.

**Qualifizierung**

`WriteLine()` ist eine Methode, die in der Klasse `Console` deklariert ist. Um dem Compiler nun mitzuteilen, dass er diese Methode dieser Klasse verwenden soll, müssen wir den Klassennamen mit angeben. Klassenname und Methodenname werden durch einen Punkt getrennt. Anders als bei verschiedenen weiteren Programmiersprachen ist der Punkt in C# der einzige Operator zur Qualifizierung von Bezeichnern – genau so nennt man nämlich diese Vorgehensweise. Man teilt dem Compiler im Prinzip genau mit, wo er die gewünschte Methode findet.

An sich ist das nicht schwer zu verstehen. Nehmen Sie nur einmal an, jemand stellt Ihnen die Frage, wer dieses Buch geschrieben hat, weil er sich über den Autor erkundigen möchte. Die Aussage „Frank“ würde dieser Person nicht ausreichen, denn es gibt sicherlich noch mehr „Franks“, die ein Buch geschrieben haben. Daher müssen Sie den Namen qualifizieren, in diesem Fall, indem Sie den Nachnamen hinzufügen. Dann hat Ihr Kollege auch eine reelle Chance, etwas über mich zu erfahren.

Auch hier wieder der Hinweis, dass Sie peinlich genau darauf achten müssen, wie Sie die Anweisungen schreiben. Die Anweisung heißt `WriteLine()`, nicht `Writeline()`. Achten Sie also immer auf die Groß- und Kleinschreibung, da der Compiler es auch tut und sich beim geringsten Fehler beschwert.

C# achtet auf die Groß- und Kleinschreibung. Im Fachjargon bezeichnet man eine solche Sprache als *Case-sensitive*. Achten Sie also darauf, wie Sie Ihre Anweisungen, Variablen, Methodenbezeichner schreiben. Die Variablen `meinWert` und `meinwert` werden als unterschiedliche Variablen behandelt.

**Semikolon**

Das Semikolon hinter der Anweisung `WriteLine()` ist ebenfalls ein wichtiger Bestandteil eines C#-Quelltextes. Anweisungen werden immer durch ein Semikolon abgeschlossen, d.h. dieses Zeichen ist ein Trennzeichen.

Damit wäre es prinzipiell möglich, mehrere Anweisungen hintereinander zu schreiben und sie einfach durch das Semikolon zu trennen. Aus Gründen der Übersichtlichkeit wird das aber nicht ausgenutzt, stattdessen verwendet man in der Regel für jede neue Anweisung auch eine neue Zeile.

Andersrum ist es aber durchaus möglich, ein langes Kommando zu trennen (nur eben nicht mitten in einem Wort) und es auf mehrere Zeilen zu verteilen. Das kann sehr zu einer besseren Übersicht beitragen, und aufgrund der maximalen Zeilenlänge in diesem Buch wurde es auch hier des Öfteren praktiziert. *Visual Basic* ist beispielsweise eine Sprache, bei der im Gegensatz dazu alle Teile einer Anweisung in einer Zeile stehen sollten. Falls dies nicht möglich ist, muss der Unterstrich als Verbindungszeichen zur nächsten Zeile verwendet werden. In C# bezeichnet das Semikolon das Ende einer Anweisung, ein Zeilenumbruch hat keinerlei Auswirkungen und ein Verbindungszeichen wie der Unterstrich ist auch nicht notwendig.

### 2.2.5 Namensräume (Namespaces)

Am Anfang des Programms sehen Sie eine Anweisung, die eigentlich gar keine Anweisung im herkömmlichen Sinne ist, sondern vielmehr als Information für den Compiler dient. Die erste Zeile lautet

```
namespace HalloWelt
```

und wir können erkennen, dass gleich danach wieder durch die geschweiften Klammern ein Block definiert wird. Wir sehen weiterhin, dass die Klasse `HalloWelt1` offensichtlich einen Bestandteil dieses Blocks darstellt, wir wissen jedoch nicht, was es mit dem reservierten Wort `namespace` auf sich hat.

**namespace**

Das reservierte Wort `namespace` bezeichnet einen so genannten Namensraum, wobei es sich um eine Möglichkeit der Untergliederung eines Programms handelt. Ein Namensraum ist ein Bereich, in dem Klassen thematisch geordnet zusammengefasst werden können, wobei dieser Namensraum nicht auf eine Datei beschränkt ist, sondern vielmehr dateiübergreifend funktioniert.

Mit Hilfe von Namensräumen können Sie eigene Klassen, die Sie in anderen Applikationen wiederverwenden wollen, thematisch grup-

pieren. So könnte man z.B. alle Beispielklassen dieses Buchs in einen Namensraum `CSharpLernen` platzieren. Später werden wir sehen, dass Namensräume auch verschachtelt werden können, also auch ein Namensraum wie `CSharpLernen.Kapitel1` möglich ist.

Im Buch sind die Programme so klein, dass nicht immer ein Namensraum angegeben ist. In eigenen, vor allem in umfangreicheren Projekten sollten Sie aber intensiven Gebrauch von dieser Möglichkeit machen.

Das .net-Framework stellt bereits eine Reihe von Namensräumen mit vordefinierten Klassen bzw. Datentypen zur Verfügung. Um die darin deklarierten Klassen zu verwenden, muss der entsprechende Namensraum aber zunächst in Ihre eigene Applikation eingebunden werden, d.h. wir teilen dem Programm mit, dass es diesen Namensraum verwenden und uns den Zugang zu den darin enthaltenen Klassen ermöglichen soll.

***using***

Das Einbinden eines Namensraums geschieht durch das reservierte Wort **`using`**. Hiermit teilen wir dem Compiler mit, welche Namensräume wir in unserem Programm verwenden wollen. Einer dieser Namensräume, den wir in jedem Programm verwenden werden, ist der Namensraum `System`. In diesem ist unter anderem auch die Klasse Console deklariert, deren Methode `WriteLine()` wir ja bereits verwendet haben. Außerdem enthält `System` die Deklarationen aller Basis-Datentypen von C#.

Wir sind jedoch nicht gezwungen, einen Namensraum einzubinden. Das Konzept von C# ermöglicht es auch, auf die in einem Namensraum enthaltenen Klassen durch die Angabe des Namensraum-Bezeichners zuzugreifen. Wenn es nur darum geht, eine Klasse oder Methode ein einziges Mal anzuwenden, kann diese Vorgehensweise durchaus einmal Verwendung finden. Würden wir beispielsweise in unserem *Hallo-Welt*-Programm die **`using`**-Direktive weglassen, müssten wir den Aufruf der Methode `WriteLine()` folgendermaßen programmieren:

```
System.Console.WriteLine("Hallo Welt");
```

Der Namensraum, in dem die zu verwendende Klasse deklariert ist, muss also mit angegeben werden. Wenn wir an dieser Stelle wieder das Beispiel mit der Person heranziehen, die Sie nach dem Namen des Autors fragt, würden Sie vermutlich zu meinem Namen noch den Hinweis hinzufügen, dass ich Autor für Addison-Wesley bin – also die Aussage präzisieren.

**Der globale Namensraum**

Die Angabe eines Namensraums für Ihr eigenes Programm ist nicht zwingend notwendig. Wenn Sie keinen angeben, wird der so genannte globale Namensraum verwendet, was allerdings bedeutet, dass die Untergliederung Ihres Programms faktisch nicht vorhanden ist. Weitaus sinnvoller ist es, verschiedene Programmteile in Namensräume zu verpacken und diese dort mittels `using` einzubinden, wo sie gebraucht werden.

Auf Namensräume und ihre Deklaration werden wir in Kapitel 3.4 nochmals genauer eingehen.

## 2.3 Hallo Welt die Zweite

Wir wollen unser kleines Programm ein wenig umbauen, so dass es einen Namen ausgibt. Der Name soll vorher eingelesen werden, wir benötigen also eine Anweisung, mit der wir eine Eingabe des Benutzers empfangen können. Diese Anweisung heißt `ReadLine()` und ist ebenfalls in der Klasse `Console` deklariert. `ReadLine()` hat keine Übergabeparameter und liefert lediglich die Eingabe des Benutzers zurück, der Aufruf gestaltet sich also wie eine Zuweisung. Aber obwohl keine Parameter an `ReadLine()` übergeben werden, müssen dennoch die runden Klammern geschrieben werden, die anzeigen, dass es sich um eine Methode handelt. Im Beispielcode werden Sie sehen, was gemeint ist. Wir bauen unser Programm also um:

```
namespace HalloWelt
{

/* Hallo Welt Konsolenapplikation */
/* Autor:   Frank Eller           */
/* Sprache: C#                    */

  using System;

  public class HalloWelt2
  {
    public static int Main(string[] args)
    {
      string theName;
      theName = Console.ReadLine();
      Console.WriteLine("Hallo {0}.", theName);
      return 0;
    }
  }
}
```

Wenn Sie eine Methode aufrufen, der keine Parameter übergeben werden, müssen Sie dennoch die runden Klammern schreiben, die anzeigen, dass es sich um eine Methode und nicht um eine Variable handelt. Auch bei der Deklaration einer solchen Methode werden die runden Klammern geschrieben, obwohl eigentlich keine Parameter übergeben werden.

An diesem Beispiel sehen Sie im Vergleich zum ersten Programm einige Unterschiede. Zum Ersten sind zwei Anweisungen hinzugekommen, zum Zweiten hat sich die Ausgabeanweisung verändert. Am übrigen Programm wurden keine Änderungen vorgenommen. Wenn Sie dieses Programm abspeichern und compilieren (Sie können auch das vorherige Programm einfach abändern), erhalten Sie zunächst eine Eingabeaufforderung. Wenn Sie nun Ihren Namen eingeben (in meinem Fall „Frank Eller"), erhalten Sie die Ausgabe

**Hallo Frank Eller.**

Sie finden auch dieses Programm auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_2\HALLOWELT2.

### 2.3.1 Variablendeklaration

*string*

Um den Namen einlesen und danach wieder ausgeben zu können, benötigen wir einen Platz, wo wir ihn ablegen können. In unserem Fall handelt es sich dabei um die Variable `theName`, die den Datentyp `string` hat. Der Datentyp `string` bezeichnet Zeichenketten im *Unicode*-Format, d.h. jedes Zeichen wird mit zwei Bit dargestellt – dadurch ergibt sich eine Kapazität des Zeichensatzes von 65535 Zeichen, was genügend Platz für jedes mögliche Zeichen aller weltweit bekannten Schriften ist. Genauer gesagt, ungefähr ein Drittel des verfügbaren Platzes ist sogar noch frei.

*Variablendeklaration*

Eine Variable wird deklariert, indem man zunächst den Datentyp angibt und dann den Bezeichner. In unserem Fall also den Datentyp `string` und den Bezeichner `theName`. Dieser Variable weisen wir die Eingabe des Anwenders zu, die uns von der Methode `Console.ReadLine()` zurückgeliefert wird. Bei diesem Wert handelt es sich ebenfalls um einen Wert mit dem Datentyp `string`, die Zuweisung ist also ohne weitere Formalitäten möglich.

Wäre der Datentyp unserer Variable ein anderer, müssten wir eine Umwandlung vornehmen, da die Methode `ReadLine()` stets eine Zeichenkette (also den Datentyp `string`) zurückliefert. Welche Möglichkeiten uns hierfür zur Verfügung stehen, werden wir in Kapitel 4.3 noch behandeln.

**Aufrufarten**

Auffallen sollte weiterhin, dass die Methode `WriteLine()` einfach so hingeschrieben wurde, die Methode `ReadLine()` aber wie eine Zuweisung benutzt wurde. Das liegt daran, dass es sich bei `ReadLine()` um eine Methode handelt, die einen Wert zurückliefert, `WriteLine()` tut dies nicht. Und diesen Wert müssen wir, wollen wir ihn verwenden, zunächst einer Variablen zuweisen. Also rufen wir die Methode `ReadLine()` so auf, als ob es sich um eine Zuweisung handeln würde.

Diese Art des Aufrufs ist nicht zwingend erforderlich. Man könnte die Methode `ReadLine()` auch dazu verwenden, den Anwender die Taste ↵ betätigen zu lassen. Dazu müsste die Eingabe des Anwenders nicht ausgewertet werden, denn im Prinzip wollen wir ja nichts damit tun – wir wollen nur, dass der Anwender ↵ drückt. In einem solchen Fall, wo das Ergebnis einer Methode keine Bedeutung für den weiteren Programmablauf hat, kann die Methode auch einfach durch Angabe des Methodennamens aufgerufen werden, wobei der Ergebniswert allerdings verworfen wird.

Methoden, die einen Wert zurückliefern, werden normalerweise wie eine Zuweisung verwendet. Im Prinzip verhalten sie sich wie Variablen, nur dass der Wert eben berechnet oder innerhalb der Methode erzeugt wird. Methoden mit dem Ergebnistyp `void` liefern keinen Wert zurück, werden also einfach nur mit ihrem Namen aufgerufen.

Wenn eine Methode, die einen Wert zurückliefert, nur durch Angabe ihres Namens aufgerufen wird, wird der Ergebniswert verworfen. Der Methodenaufruf an sich funktioniert aber dann auch.

**lokale Variablen**

Die von uns deklarierte Variable `theName` hat noch eine weitere Besonderheit. Da sie innerhalb des Anweisungsblocks der Methode `Main()` deklariert wurde, ist sie auch nur dort gültig. Man sagt, es handelt sich um eine *lokale Variable*. Wenn eine Variable innerhalb eines durch geschweifte Klammern bezeichneten Blocks deklariert wird, ist sie auch nur dort gültig und nur so lange existent, wie der Block abgearbeitet wird. Variablen, die innerhalb eines Blocks deklariert werden, sind immer lokal für den Block gültig, in dem sie deklariert sind.

Variablen, die innerhalb eines Blocks deklariert sind, sind auch nur innerhalb dieses Blocks gültig. Man bezeichnet sie als *lokale Variablen*. Von außerhalb kann auf diese Variablen bzw. ihre Werte nicht zugegriffen werden.

### 2.3.2 Die Platzhalter

An unserer Ausgabeanweisung hat sich ebenfalls etwas geändert. Vergleichen wir kurz „vorher“ und „nachher“. Die Anweisung

```
Console.WriteLine("Hallo Welt");
```

hat sich geändert zu

```
Console.WriteLine("Hallo {0}.", theName);
```

*Platzhalter*

Der Ausdruck `{0}` ist ein so genannter Platzhalter für einen Wert. Der eigentliche Wert, der ausgegeben werden soll, wird nach der auszugebenen Zeichenkette angegeben, in unserem Fall handelt es sich um die Variable `theName` vom Typ **`string`**. Die Methode `WriteLine()` kann dabei alle Datentypen verarbeiten.

Es ist auch möglich, mehr als einen Wert anzugeben. Dann werden mehrere Platzhalter benutzt (für jeden auszugebenden Wert einer) und durchnummeriert, und zwar wie fast immer bei Programmiersprachen mit dem Wert 0 beginnend. Bei drei Werten, die ausgegeben werden sollen, also `{0}`, `{1}` und `{2}`.

*object*

Der Datentyp der Parameter ist **`object`**, die Basisklasse aller Klassen in C#. Damit ist es möglich, als Parameter jeden Datentyp zu verwenden, also sowohl Zeichenketten, ganze Zahlen, reelle Zahlen usw. `WriteLine()` konvertiert die Daten automatisch in den richtigen Datentyp für die Ausgabe. Wie das genau funktioniert und wie Sie die Ausgabe von Zahlenwerten auch selbst formatieren können, erfahren Sie noch im weiteren Verlauf des Buchs.

### 2.3.3 Escape-Sequenzen

Wir haben bereits etwas ausgegeben, bisher allerdings nur den Satz „Hallo Welt“. Es gibt aber noch weitere Möglichkeiten, Dinge auszugeben und auch diese Ausgabe zu formatieren. Dazu verwendet man Sonderzeichen, so genannte *Escape-Sequenzen*.

Früher wurden diese Escape-Sequenzen für Drucker im Textmodus benutzt, um diesen anzuzeigen, dass statt eines Zeichens jetzt ein Befehl folgt. Man hat dafür den ASCII-Code der Taste Esc verwendet, daher auch der Name *Escape-Sequenz*.

Die Ausgabe im Textmodus geschieht über die Methoden `Write()` oder `WriteLine()` der Klasse `Console`. Wir wissen bereits, dass es sich um statische Methoden handelt, wir also keine Instanz der Klasse `Console` erzeugen müssen. Die Angabe der auszugebenden Zeichenkette erfolgt in Anführungszeichen, und innerhalb dieser Anführungszeichen

können wir nun solche Escape-Sequenzen benutzen, um die Ausgabe zu manipulieren.

Der Unterschied zwischen `Write()` und `WriteLine()` besteht lediglich darin, dass `WriteLine()` an die Ausgabe noch einen Zeilenvorschub anhängt, `Write()` nicht. Wenn Sie also wie in unserem Fall eine Aufforderung zur Eingabe programmieren wollen, bei der der Anwender seine Eingabe direkt hinter der Aufforderung machen kann, benutzen Sie `Write()`.

***Backslash ( \ )***

Alle Escape-Sequenzen werden eingeleitet durch einen *Backslash*, einen rückwärtigen Schrägstrich, das Trennzeichen für Verzeichnisse unter Windows. Tabelle 2.4 zeigt die Escape-Zeichen von C# in der Übersicht.

| Zeichen | Bedeutung | Unicode |
|---|---|---|
| \a | Alarm – Wenn Sie dieses Zeichen ausgeben, wird ein Signalton ausgegeben. | 0007 |
| \t | Entspricht der Taste ⇆ | 0009 |
| \r | Entspricht einem Wagenrücklauf, also der Taste ↵ | 000A |
| \v | Entspricht einem vertikalen Tabulator | 000B |
| \f | Entspricht einem *Form Feed*, also einem Seitenvorschub | 000C |
| \n | Entspricht einer neuen Zeile | 000D |
| \e | Entspricht der Taste Esc | 001B |
| \c | Entspricht einem ASCII-Zeichen mit der Strg-Taste, also entspricht \cV der Tastenkombination Strg + V | |
| \x | Entspricht einem ASCII-Zeichen. Die Angabe erfolgt allerdings als Hexadezimal-Wert mit genau zwei Zeichen. | |
| \u | Entspricht einem Unicode-Zeichen. Sie können einen 16-Bit-Wert angeben, das entsprechende Unicode-Zeichen wird dann ausgegeben. | |
| \ | Wenn hinter dem Backslash kein spezielles Zeichen steht, das der Compiler erkennt, wird das Zeichen ausgegeben, das direkt dahinter steht. | |

*Tabelle 2.1: Ausgabe spezieller Zeichen*

Vor allem die letzte Zeile mag etwas Verwirrung stiften, denn wozu sollte man den Backslash vor ein Zeichen setzen, wenn dieser ohnehin nur bewirkt, dass genau dieses Zeichen ausgegeben wird? Der Grund ist ganz einfach. Es gibt Zeichen, die der Compiler erkennt und die eine bestimmte Bedeutung haben. Das einfachste Beispiel ist das Anführungszeichen, das im Programmcode für den Beginn einer Zeichenkette steht. Wie also sollte man dieses Zeichen ausgeben?

Nun, einfach über eine Escape-Sequenz, also mit vorangestelltem Backslash:

```
/* Beispiel Escape-Sequenzen */
/* Autor:   Frank Eller      */
/* Sprache: C#               */

using System;

class TestClass
{
  public static void Main()
  {
    Console.WriteLine("Ausgabe mit \"Anführungszeichen\"");
  }
}
```

Wenn Sie das obige Beispiel eingeben und ausführen, ergibt sich folgende Ausgabe:

**Ausgabe mit "Anführungszeichen"**

Sie finden das Programm auch auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_2\ESC_SEQ.

## 2.4 Zusammenfassung

Dieses Kapitel war lediglich eine Einführung in die große Welt der C#-Programmierung. Sie können aber bereits erkennen, dass es nicht besonders schwierig ist, gleich schon ein Programm zu schreiben. Die wichtigste Methode eines Programms – die Methode `Main()` – haben Sie nun kennen gelernt, auch dass ein C#-Programm unbedingt eine Klasse benötigt, wurde angesprochen.

Klar ist, dass die Informationen, die Sie aus einem solchen Kapitel mitnehmen können, noch sehr vage sind. Es wäre ja auch schlimm für die Programmiersprache, wenn so wenig dazu nötig wäre, sie zu erlernen. Dann wären nämlich auch die Möglichkeiten recht eingeschränkt. C# ist jedoch eine Sprache, die sehr umfangreiche Möglichkeiten bietet und im Verhältnis dazu leicht erlernbar ist.

Wir werden in den nächsten Kapiteln Stück für Stück tiefer in die Materie einsteigen, so dass Sie am Ende des Buchs einen besseren Überblick über die Möglichkeiten von C# und die Art der Programmierung mit dieser neuen Programmiersprache erhalten haben. Das soll nicht bedeuten, dass Sie lediglich eine oberflächliche Betrachtung erhalten haben, Sie werden sehr wohl in der Lage sein, eigene Pro-

gramm zu schreiben. Dieses Buch dient dazu, die Basis für eine erfolgreiche Programmierung zu legen.

## 2.5 Kontrollfragen

Kontrollieren Sie sich selbst. Am Ende eines Kapitels werden Sie immer einige Kontrollfragen und/oder auch einige Übungen finden, die Sie durchführen können. Diese dienen dazu, Ihr Wissen sowohl zu kontrollieren als auch zu vertiefen. Sie sollten die Übungen und die Kontrollfragen daher immer sorgfältig durcharbeiten. Die Lösungen finden Sie in Kapitel 12. Und hier bereits einige Fragen zum Kapitel *Erste Schritte*:

1. Warum ist die Methode `Main()` so wichtig für ein Programm?
2. Was bedeutet das Wort `public`?
3. Was bedeutet das Wort `static`?
4. Welche Arten von Kommentaren gibt es?
5. Was bedeutet das reservierte Wort `void`?
6. Wozu dient die Methode `ReadLine()`?
7. Wie kann ich einen Wert oder eine Zeichenkette ausgeben?
8. Was bedeutet `{0}`?
9. Was ist eine lokale Variable?
10. Wozu werden Escape-Sequenzen benötigt?

# 3 Programmstrukturierung

In diesem Kapitel werden wir uns mit dem grundsätzlichen Aufbau eines C#-Programms beschäftigen. Sie werden einiges über Klassen und Objekte erfahren, die die Basis eines jeden C#-Programms darstellen, weitere Informationen über Namensräume erhalten und über die Deklaration sowohl von Variablen als auch von Methoden einer Klasse.

C# bietet einige Möglichkeiten der Strukturierung eines Programms, allen voran natürlich das Verpacken der Funktionalität in Klassen. So können Sie mehrere verschiedene Klassen erstellen, die jede für sich in ihrem Bereich eine Basisfunktionalität bereitstellt. Zusammengenommen entsteht aus diesen einzelnen Klassen ein Gefüge, das fertige Programm mit erweiterter Funktionalität, indem die einzelnen Klassen miteinander interagieren und jede genau die Funktionalität bereitstellt, für die sie programmiert wurde.

## 3.1 Klassen und Objekte

### 3.1.1 Deklaration von Klassen

Klassen sind eigentlich abstrakte Gebilde, die nicht direkt verwendet werden können. Stattdessen müssen von den Klassen so genannte *Objekte* erzeugt werden, die dann im Programm verwendet werden können. Man sagt auch, es wird eine *Instanz* einer Klasse erzeugt, die angesprochenen Objekte sind also nichts weiter als Instanzen von Klassen.

***dynamischer Speicher***

Wenn ein Objekt erzeugt wird, wird dynamisch Speicher für dieses Objekt reserviert, der irgendwann auch wieder freigegeben werden muss. Sinnvollerweise sollte das in dem Moment geschehen, in dem das Objekt nicht mehr benötigt wird. In anderen Programmierspra-

chen kam es deswegen immer wieder zu dem Problem der so genannten „Speicherleichen“. Dabei wurden zwar Instanzen von Klassen erzeugt, aber nicht wieder freigegeben. Das Problem war, dass der reservierte Speicher auch dann noch reserviert (belegt) blieb, nachdem das Programm bereits beendet war. Erst durch einen Neustart des Systems wurde er wieder freigegeben. Vor allem C++-Programmierer kennen diese Problematik.

**Garbage Collection**

C# bzw. das .net-Framework nehmen Ihnen diese Aufgabe ab. Sobald ein Objekt nicht mehr benötigt wird, wird der Speicher, den es belegt, automatisch wieder freigegeben. Verantwortlich dafür ist die so genannte *Garbage-Collection*, eine Art Müllabfuhr im Hauptspeicher. Diese kümmert sich automatisch darum, dass dynamisch belegter Speicher wieder freigegeben wird, Sie selbst müssen sich nicht darum kümmern.

**Klassendeklaration**

Doch zurück zu unseren Klassen. Am Anfang steht also die Deklaration der Klasse mit ihren Eigenschaften und der Funktionalität. Die Eigenschaften werden in so genannten *Feldern*, die Funktionalität in den *Methoden* der Klasse zur Verfügung gestellt. Abbildung 3.1 zeigt den Zusammenhang.

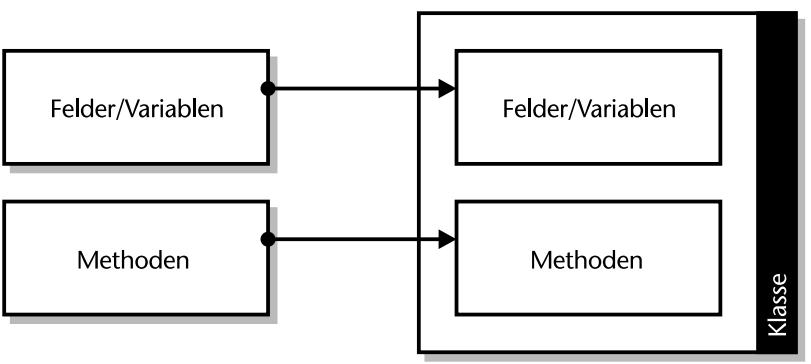


*Abbildung 3.1: Struktur einer Klasse*

In der Abbildung wird die Vermutung nahe gelegt, dass Felder und Methoden getrennt, zuerst die Felder und dann die Methoden, deklariert werden müssen. Das ist aber nicht der Fall. Es muss lediglich sichergestellt werden, dass ein Feld vor seiner erstmaligen Verwendung deklariert und initialisiert ist, d.h. einen Wert enthält. Ist dies nicht der Fall, meldet sich der Compiler mit einem Fehler.

**Attribute**

Beides zusammen, Felder und Methoden, nennt man auch die *Attribute* einer Klasse. Der Originalbegriff, der ebenfalls häufig verwendet wird, ist *Member* (engl., *Mitglied*). In diesem Buch werde ich aber den Begriff *Attribut* benutzen.

Manche Programmierer bezeichnen auch gerne nur die Felder als Attribute und die Methoden als Methoden. Lassen Sie sich dadurch nicht verwirren, normalerweise wird im weiteren Verlauf einer Unterhaltung klar, was genau gemeint ist. Wenn in diesem Buch von Attributen gesprochen wird, so handelt es sich immer um beides, Felder und Methoden einer Klasse.

### 3.1.2 Erzeugen von Instanzen

Möglicherweise werden Sie sich fragen, wozu die Erstellung einer Instanz dient, macht sie doch die ganze Programmierung ein wenig komplizierter. Nun, so kompliziert, wie es aussieht, wird die Programmierung dadurch aber nicht, und außerdem macht es durchaus Sinn, dass von einer Klasse zunächst eine Instanz erzeugt werden muss.

Nehmen wir an, Sie hätten eine Klasse `Fahrzeug` deklariert, die Sie nun verwenden wollen. Ein Fahrzeug kann vieles sein, z.B. ein Auto, ein Motorrad oder ein Fahrrad. Wenn Sie nun die Klasse direkt verwenden könnten, müssten Sie entweder alle diese Fahrzeuge in der Klassendeklaration berücksichtigen oder aber für jedes Fahrzeug eine neue Klasse erzeugen, wobei die verwendeten Felder und Methoden zum größten Teil gleich wären (z.B. Beschleunigen oder Bremsen, nur um ein Beispiel zu nennen).

***Instanzen***

Stattdessen verwenden wir nur eine Basisklasse und erzeugen für jedes benötigte Fahrzeug eine Instanz. Diese Instanz können Sie sich wie eine Kopie der Klasse im Speicher vorstellen, wobei Sie in der Klasse alle Basisinformationen deklarieren, die Werte aber der erzeugten Instanz respektive dem erzeugten Objekt zuweisen. Sie haben also die Möglichkeit, obwohl nur eine einzige Klasse existiert, mehrere Objekte mit unterschiedlichem Verhalten davon abzuleiten.

***new***

Die Erzeugung einer Klasse geschieht in C# mit dem reservierten Wort **new**. Dieses Wörtchen bedeutet für den Compiler „erzeuge eine neue Kopie des nachfolgenden Datentyps im Speicher des Computers“. Um also die angegebenen Objekte aus einer Klasse `Fahrzeug` zu erzeugen, wären folgende Anweisungen notwendig:

```
Fahrzeug Fahrrad  = new Fahrzeug();
Fahrzeug Motorrad = new Fahrzeug();
Fahrzeug Auto     = new Fahrzeug();
```

Die Klasse `Fahrzeug` ist dabei die Basis. Die erzeugten Objekte werden später innerhalb des Programms benutzt, enthalten die Funktionalität, die in der Klasse deklariert wurde, sind aber ansonsten eigenstän-

dig. Sie können diesen Objekten dann die unterschiedlichen Eigenschaften zuweisen und somit ihr Verhalten beeinflussen. Abbildung 3.2 zeigt schematisch, wie die Instanziierung funktioniert.

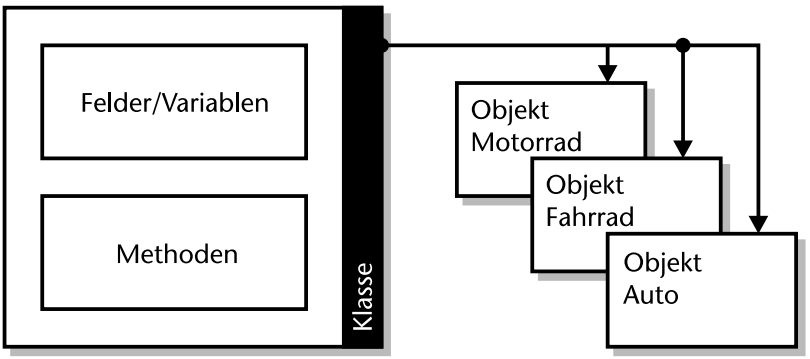


*Abbildung 3.2: Objekte (Instanzen) aus einer Klasse erzeugen*

Die Instanz einer Klasse ist ein Objekt. Ebenso ist ein Objekt eine Instanz einer Klasse. In diesem Buch werden beide Wörter benutzt, auch aus dem Grund, weil es eigentlich keine weiteren Synonyme für diese Begriffe gibt. Es wäre ziemlich schlecht lesbar, wenn in einem Satz dreimal das Wort Instanz auftauchen würde.

## 3.2 Felder einer Klasse

### 3.2.1 Deklaration von Feldern

Die Eigenschaften einer Klasse sind in den Datenfeldern gespeichert. Diese Datenfelder sind nichts anderes als Variablen oder Konstanten, die innerhalb der Klasse deklariert werden und auf die dann zugegriffen werden kann. Die Deklaration einer Variablen wird folgendermaßen durchgeführt:

***Syntax***

```
[Modifikator] Datentyp Bezeichner [= Initialwert];
```

Auf den Modifikator, für Sichtbarkeit und Verhalten der Variable zuständig, kommen wir später noch zu sprechen. Für die ersten Deklarationen können Sie ihn noch weglassen.

In der Klassendeklaration wird nicht nur vorgegeben, welchen Datentyp die Felder besitzen, es kann ihnen auch ein Initialwert zugewiesen werden. Die eigentliche Zuweisung geschieht aber mit Hilfe der erzeugten Objekte (außer bei statischen Variablen/Feldern, auf

die wir später noch zu sprechen kommen). Der Initialwert ist deshalb wichtig, weil eine Variable in C# vor ihrer ersten Verwendung immer initialisiert werden muss.

Im Buch wird des Öfteren von Variablen gesprochen, auch wenn Felder gemeint sind. Das erscheint inkonsequent, ist aber eigentlich korrekt. Es gibt in C# drei verschiedene Arten von Variablen. Das sind einmal die Instanzvariablen (oder Felder), dann die statischen Variablen (oder statischen Felder) und dann noch die lokalen Variablen, die keine Felder sind. Bei einem Feld handelt es sich also um nichts anderes als um eine Variable.

***int***

Variablen bestehen aus einem Bezeichner, durch den sie innerhalb des Programms angesprochen werden können, und aus einem Datentyp, der angibt, welche Art von Information in dem entsprechenden Feld gespeichert werden kann. Einer der einfachsten Datentypen ist der *Integer*-Datentyp, ein ganzzahliger Typ mit Vorzeichen. Die Deklaration einer Variable mit dem Datentyp *Integer* geschieht in C# über das reservierte Wort `int`:

```
int myInteger;
```

***Groß-/ Kleinschreibung***

Behalten Sie dabei immer im Hinterkopf, dass C# auf die Groß- und Kleinschreibung achtet. Sie müssen daher darauf achten, dass Sie den Bezeichner bei seiner späteren Verwendung wieder genauso schreiben, wie Sie es bei der Deklaration getan haben.

***Int32***

Der Datentyp `int` ist ein Alias für den im Namensraum `System` deklarierten Datentyp `Int32`. Alle Basisdatentypen sind in diesem Namensraum deklariert, für die am häufigsten verwendeten Datentypen existieren aber Aliase, damit sie leichter ansprechbar sind.

***Initialisierung***

Wie oben angesprochen muss eine Variable (bzw. ein Feld) vor ihrer ersten Verwendung initialisiert werden, d.h. wir müssen ihr einen Wert zuweisen. Dabei gilt die erste Zuweisung als Initialisierung, was Sie entweder zur Laufzeit des Programms innerhalb einer Methode oder bereits bei der Deklaration der Variablen erledigen können. Zu beachten ist dabei, dass Sie die Variable nicht benutzen dürfen, bevor sie initialisiert ist, was in einem Fehler resultieren würde. Ein Beispiel soll dies verdeutlichen:

```
/* Beispielprogramm Initialisierung Variablen */
/* Autor:   Frank Eller                       */
/* Sprache: C#                                */

class TestClass
{
  int myNumber;  //nicht initialisiertes Feld
  int theNumber; //nicht initialisiertes Feld

  public static void Main()
  {
    myNumber = 15;  //Erste Zuweisung = Initialisierung
    myNumber = theNumber  // FEHLER: theNumber nicht
                          // initialisiert!!
  }
}
```

Wie das Schema der Deklarationsanweisung bereits zeigt, können Sie Deklaration und Initialisierung auch zusammenfassen:

```
int myInteger = 5;
```

Diese Vorgehensweise hat den Vorteil, dass keine Variable mit willkürlichen Werten belegt ist. In C# muss jede Variable vor ihrer ersten Verwendung initialisiert worden sein, es ist allerdings unerheblich, wo dies geschieht. Wichtig ist wie gesagt, dass es vor der ersten Verwendung geschieht.

### 3.2.2 Bezeichner und Schreibweisen

Wir haben im vorhergegangenen Abschnitt bereits einen Bezeichner verwendet. Es gibt allerdings noch einige Regeln, die Sie beachten sollten, weil sie dazu beitragen, die Programmierung auch größerer Applikationen zu vereinfachen. Es geht dabei schlicht um die Bezeichner und die Schreibweisen, deren sinnvoller Einsatz eine große Arbeitserleichterung mit sich bringen kann.

***sinnvolle Bezeichner***

Die erste Regel lautet: Benutzen Sie sinnvolle Bezeichner. Variablennamen wie `x` oder `y` können sinnvoll bei Koordinaten sein, in den meisten Fällen werden Sie aber aussagekräftigere Bezeichner benötigen. Gute Beispiele sind `myString`, `theName`, `theResult` usw. Schlechte Beispiele wären z.B. `x1`, `y332`, `_h5`. Diese Namen sagen absolut nichts aus, außerdem machen sie eine spätere Wartung schwierig.

Stellen Sie sich vor, Sie sollten ein Programm, das Sie nicht selbst geschrieben haben, mit einigen Funktionen erweitern. Normalerweise kein Problem. Es wird aber ein Problem, wenn der vorherige Program-

mierer nicht mit eindeutigen Bezeichnern gearbeitet hat. Gleiches gilt für Ihre eigenen Programme, wenn sie nach längerer Zeit nochmals Modifikationen vornehmen müssen. Auch dies kann zu einer schweißtreibenden Arbeit werden, wenn Sie Bezeichner verwendet haben, die keine klare Aussage über ihren Verwendungszweck machen. Glauben Sie mir, wenn ich Ihnen sage, dass es ohnehin schon schwierig genug ist.

***gleiche Schreibweise***

Eindeutige Bezeichner sind eine Sache, die nächste Regel ergibt sich aus der Tatsache, dass C# Groß- und Kleinschreibung unterscheidet. Um sicher zu sein, dass Sie Ihre Bezeichner auch über das gesamte Programm hinweg immer gleich schreiben, suchen Sie sich eine Schreibweise aus und bleiben Sie dabei, was immer auch geschieht. Mit C# ist Microsoft von der lange benutzten ungarischen Notation für Variablen und Methoden abgekommen, die ohnehin am Schluss jeden Programmierer unter Windows eher verwirrt hat statt ihm zu helfen (was eigentlich der Sinn einer einheitlichen Notation ist). Grundsätzlich haben sich nur zwei Schreibweisen durchgesetzt, nämlich das so genannte *PascalCasing* und das *camelCasing*.

***PascalCasing***

Beim *PascalCasing* wird, wie man am Namen auch schon sieht, der erste Buchstabe großgeschrieben. Weiterhin wird jeweils der erste Buchstabe eines Wortes innerhalb des Bezeichners ebenfalls wieder groß geschrieben, wodurch sich eine gute Lesbarkeit ergibt, obwohl die Wörter aneinander geschrieben sind.

***camelCasing***

Beim *camelCasing* wird der erste Buchstabe des Bezeichners kleingeschrieben. Ansonsten funktioniert es wie das *PascalCasing*, jedes weitere auftauchende Wort innerhalb des Bezeichners wird wieder mit einem Großbuchstaben begonnen.

Innerhalb dieses Buchs sind beide Schreibweisen etwas vermischt, was sich daraus ergibt, dass ich selbst natürlich auch eine (für mich klare und einheitliche) Namensgebung verwende. So deklariere ich z.B. lokale Variablen (wir werden diese später noch durchnehmen) mit Hilfe des *camelCasing*, Methoden und Felder aber mit Hilfe des *PascalCasing*. Das hat natürlich seinen Grund, unter anderem den, dass beim Aufruf einer Methode der Objektname und der Methodenbezeichner durch einen Punkt getrennt werden. Durch die obige Konvention ist sichergestellt, dass nach einem Punkt mit einem Großbuchstaben weitergeschrieben wird.

Im Falle von Eigenschaften, so genannten *Properties*, gehe ich anders vor und deklariere die Felder dann mit *camelCasing*, während ich die eigentliche Eigenschaft mit *PascalCasing* deklariere. Als Programmierer greift man später nur noch auf die Eigenschaft zu, da das verwen-

dete Feld nicht erreichbar ist, also ist wiederum sichergestellt, dass nach dem Punkt für die Qualifizierung mit einem Großbuchstaben begonnen wird.

In diesem Abschnitt wurde das Wort Eigenschaft so benutzt, als sei es ein Bestandteil einer Klasse. Wir haben weiter oben allerdings gesagt, dass die Eigenschaften einer Klasse in ihren Feldern gespeichert werden.

Es gibt jedoch auch noch so genannte Properties, Eigenschaften, die ein Bestandteil der Klasse sind. Nichtsdestotrotz benötigen auch diese eine Variable bzw. ein Feld zur Speicherung der Daten. Mehr über Eigenschaften erfahren Sie in Kapitel 9 des Buchs.

***Regeln***

Bezeichner dürfen in C# mit einem Buchstaben oder einem Unterstrich beginnen. Innerhalb des Bezeichners dürfen Zahlen zwar auftauchen, ein Bezeichner darf aber nicht mit einer solchen beginnen. Und natürlich darf ein Bezeichner nicht aus mehreren Wörtern bestehen.

Beispiele für korrekte Bezeichner sind z.B.:

```
myName
_theName
x1
Name5S7
```

Beispiele für Bezeichner, die nicht erlaubt sind, wären unter anderem:

```
1stStart
Mein Name
&again
```

***Maximallänge***

In C++ gibt es die Beschränkung, dass Bezeichner nur anhand der ersten 31 Zeichen unterschieden werden. Diese Beschränkung gilt nicht für C#, zumindest nicht, soweit ich es bisher festgestellt habe. Sie könnten also, wenn Sie wollen, durchaus auch längere Bezeichner für Ihre Variablen benutzen. Aber ich rate Ihnen davon ab, denn Sie werden sehr schnell feststellen, dass es nichts Schlimmeres gibt, als endlos lange Bezeichner.

***reservierte Wörter als Bezeichner***

Eine Besonderheit weist C# gegenüber anderen Programmiersprachen bei der Namensvergabe für die Bezeichner noch auf. In jeder Sprache gibt es reservierte Wörter, die eine feste Bedeutung haben und für nichts anderes herangezogen werden dürfen. Das bedeutet unter anderem, dass diese reservierten Wörter auch nicht als Variab-

len- oder Methodenbezeichner fungieren können, da der Compiler sie ja intern verwendet.

In C# ist das ein wenig anders gelöst worden. Hier ist es tatsächlich möglich, die auch in C# vorhandenen reservierten Wörter als Bezeichner zu verwenden, wenn man den berühmten „Klammeraffen" davor setzt. Die folgende Deklaration ist also absolut zulässig:

```
public int @int(string @string)
{
  //Anweisungen
};
```

Diese Art der Namensvergabe hat allerdings einen großen Nachteil, nämlich die mangelnde Übersichtlichkeit des Quelltextes. Dass etwas möglich ist, bedeutet noch nicht, dass man es auch unbedingt anwenden sollte. Ich selbst bin bisher in jeder Programmiersprache ganz gut ohne die Verwendung reservierter Wörter als Bezeichner ausgekommen, und ich denke, das wird auch weiterhin so bleiben. Wenn Sie dieses Feature nutzen möchten, steht es Ihnen selbstverständlich zur Verfügung, ich selbst bin der Meinung, dass es eher unnötig ist.

### 3.2.3 Modifikatoren

Bei der Deklaration von Variablen haben wir bereits die Modifikatoren angesprochen. Mit diesen Modifikatoren haben Sie als Programmierer Einfluß auf die Sichtbarkeit und das Verhalten von Variablen, Konstanten, Methoden, Klassen oder auch anderen Objekten. Tabelle 3.1 listet zunächst die Modifikatoren von C# auf.

| Modifikator | Bedeutung |
|---|---|
| public | Auf die Variable oder Methode kann auch von außerhalb der Klasse zugegriffen werden. |
| private | Auf die Variable oder Methode kann nur von innerhalb der Klasse bzw. des Datentyps zugegriffen werden. Innerhalb von Klassen ist dies Standard. |
| internal | Der Zugriff auf die Variable bzw. Methode ist beschränkt auf das aktuelle Projekt. |
| protected | Der Zugriff auf die Variable oder Methode ist nur innerhalb der Klasse bzw. durch Klassen, die von der aktuellen Klasse abgeleitet sind, möglich. |
| abstract | Dieser Modifikator bezeichnet Klassen, von denen keine Instanz erzeugt werden kann. Von abstrakten Klassen muss immer zunächst eine Klasse abgeleitet werden. |

*Tabelle 3.1: Die Modifikatoren von C#*

| Modifikator | Bedeutung |
|---|---|
| const | Der Modifikator für Konstanten. Der Wert von Feldern, die mit diesem Modifikator deklariert wurden, ist nicht mehr veränderlich. |
| event | Deklariert ein Ereignis (engl. *Event*) |
| extern | Dieser Modifikator zeigt an, dass die entsprechend bezeichnete Methode extern (also nicht innerhalb des aktuellen Projekts) deklariert ist. Sie können so auf Methoden zugreifen, die in DLLs deklariert sind. |
| override | Dient zum Überschreiben bereits implementierter Methoden beim Ableiten einer Klasse. Sie können eine Methode, die in der Basisklasse deklariert ist, in der abgeleiteten Klasse überschreiben. |
| readonly | Mit diesem Modifikator können Sie ein Datenfeld deklarieren, dessen Werte von außerhalb der Klasse nur gelesen werden können. Innerhalb der Klasse ist es nur möglich, Werte über den Konstruktor oder direkt bei der Deklaration zuzuweisen. |
| sealed | Der Modifikator **sealed** versiegelt eine Klasse. Fortan können von dieser Klasse keine anderen Klassen mehr abgeleitet werden. |
| static | Ein Feld oder eine Methode, die als **static** deklariert ist, gilt als Bestandteil der Klasse selbst. Die Verwendung der Variable bzw. der Aufruf der Methode benötigt keine Instanziierung der Klasse. |
| virtual | Der Modifikator **virtual** ist sozusagen das Gegenstück zu override. Mit **virtual** werden die Methoden einer Klasse festgelegt, die später überschrieben werden können (mittels **override**). Mehr über **virtual** und **override** in Kapitel 8, wenn wir tiefer in die Programmierung eigener Klassen einsteigen. |

*Tabelle 3.1: Die Modifikatoren von C# (Forts.)*

Nicht alle dieser Modifikatoren sind immer sinnvoll bzw. möglich, es hängt von der Art der Deklaration und des Datentyps ab. Die Modifikatoren, die möglich sind, lassen sich dann aber auch kombinieren, so dass Sie eine Methode oder ein Datenfeld durchaus als **public** und **static** gleichzeitig deklarieren können. Gesehen haben Sie das ja bereits bei der Methode `Main()`.

Modifikatoren sind einigen Lesern möglicherweise bereits aus Java bekannt. Der Umgang damit ist nicht weiter schwer, manch einer muss sich lediglich etwas umgewöhnen. Sie werden aber im Verlauf des Buchs immer wieder davon Gebrauch machen können und mit der Zeit werden Ihnen zumindest die gebräuchlichsten Modifikatoren in Fleisch und Blut übergehen. An dieser Stelle nur noch ein paar wichtige Hinweise für Neueinsteiger:

– Die Methode `Main()` als Hauptmethode eines jeden C#-Programms wird immer mit den Modifikatoren **public** und **static** deklariert. Keine Ausnahme.
– Die möglichen Modifikatoren können miteinander kombiniert werden, es sei denn, sie würden sich widersprechen (so macht eine Deklaration mit den Modifikatoren **public** und **private** zusammen keinen Sinn ...).
– Modifikatoren stehen bei einer Deklaration immer am Anfang.

Wenn Sie diese Hinweise ein wenig im Hinterkopf behalten, wird Ihnen der Umgang mit C#-typischen Deklarationen schnell sehr leicht fallen.

***Standard-Modifikatoren***

Weiter oben wurde angegeben, dass ein Modifikator nicht unbedingt notwendig ist. Das ist so zwar richtig, zumindest aber der Sichtbarkeitsbereich wird dennoch festgelegt. Wenn ein Feld innerhalb einer Klasse deklariert wird und kein Modifikator angegeben wurde, so ist dieses Datenfeld automatisch als **private** deklariert, d.h. von außerhalb der Klasse kann nicht darauf zugegriffen werden. Wollen Sie dem Datenfeld (oder der Methode, denn für Methoden gilt das Gleiche) eine andere Sichtbarkeitsstufe zuweisen, so müssen Sie einen Modifikator benutzen.

```
/* Beispielprogramm Modifikatoren */
/* Autor:   Frank Eller           */
/* Sprache: C#                    */

public class TestClass
{
  public int myNumber = 10  //öffentlich
  int theNumber = 15;       //private
}

class TestClass2
{
  public static void Main()
  {
    TestClass myClass = new TestClass();

    TestClass.myNumber = 10; //ok, myNumber ist public
    TestClass.theNumber= 15; //FEHLER, theNumber ist private
  }
}
```

Für jede Variable, jede Methode, Klasse oder jeden selbst definierten Datentyp gilt immer genau der Modifikator, der direkt davor steht. Es ist in C# nicht möglich, einen Modifikator gleichzeitig auf mehrere Deklarationen anzuwenden. Wenn kein Modifikator verwendet wird, gilt innerhalb von Klassen der Modifikator **private**.

## 3.3 Methoden einer Klasse

### 3.3.1 Deklaration von Methoden

Methoden beinhalten die Funktionalität einer Klasse. Hierzu werden innerhalb der Methode Anweisungen verwendet, wobei es sich um Zuweisungen, Aufrufe anderer Methoden, Deklarationen, Verzweigungen oder Schleifen handeln kann. Auf die verschiedenenen Konstrukte wird im Verlauf des Buchs noch genauer eingegangen. Dieses Kapitel soll vielmehr aufzeigen, wie die Deklaration einer Methode vonstatten geht.

***Methodendeklaration***

Die Deklaration einer Methode sieht so ähnlich aus wie die Deklaration einer Variable, wobei eine Methode noch einen Programmblock beinhaltet, der die Anweisungen enthält. Weiterhin können Methoden Werte zurückliefern und auch Werte empfangen, nämlich über Parameter. Die Deklaration einer Methode hat die folgende Syntax:

***Syntax***

```
[Modifikator] Ergebnistyp Bezeichner ([Parameter])
{
  // Anweisungen
}
```

***Modifikatoren***

Für die Modifikatoren gilt das Gleiche wie für die Variablen. Wenn innerhalb einer Klasse kein Modifikator benutzt wird, gilt als Standard die Sichtbarkeitsstufe **private**. Außerdem können auch Methoden als **static** deklariert werden bzw. andere Sichtbarkeitsstufen erhalten.

***statische Methoden***

Ein Beispiel für eine öffentliche, statische Methode ist ja bereits unsere Methode `Main()`, ein weiteres Beispiel ist die Methode `WriteLine()` der Klasse `Console`. Sie werden festgestellt haben, dass wir in unserem *Hallo-Welt*-Programm keine Instanz der Klasse `Console` erstellen mussten, um die Methoden `WriteLine()` bzw. `ReadLine()` zu verwenden. Beides sind öffentliche, statische Methoden. Für den Moment müssen wir uns in diesem Zusammenhang allerdings nur merken, dass statische Methoden Bestandteil der Klassendeklaration sind, nicht des aus der Klasse erzeugten Objekts. In Kapitel 3.3.7 werden wir noch ein wenig genauer auf statische Methoden eingehen.

**Prototypen**

Aus C++ sind Ihnen möglicherweise die *Prototypen* oder die *Forward-Deklarationen* bekannt. Dabei muss eine Methode bereits vor ihrer Implementation angekündigt werden. Dazu wird der Kopf einer Methode verwendet – Er wird angegeben, die eigentliche Deklaration der Methode folgt irgendwo im weiteren Verlauf des Quelltextes. In C++ ist es so, dass diese Forward-Deklarationen bzw. Prototypen in einer so genannten Header-Datei zusammengefasst werden, während sich die Implementationen der Methoden dann in der .CPP-Datei befinden.

C# arbeitet ohne Prototypen. Es sind in dieser Sprache keinerlei Forward-Deklarationen notwendig, d.h. Sie können Ihre Methode deklarieren, wo immer Sie wollen, der Compiler wird sie finden. Natürlich müssen Sie dabei im Gültigkeitsbereich der jeweiligen Klasse bleiben. Prinzipiell aber müssen Sie lediglich gleich beim Deklarieren einer Methode auch den dazugehörigen Programmtext eingeben. C# findet die Methode dann von sich aus.

**Deklarationen mischen**

Auch kann in C# die Deklaration von Feldern und Methoden gemischt werden. Wie gesagt, es ist vollkommen unerheblich, weil der Compiler vom gesamten Gültigkeitsbereich der Klasse ausgeht. Sie müssen lediglich darauf achten, dass eine Variable deklariert und initialisiert ist, bevor Sie sie das erste Mal nutzen. Dazu ein kleines Beispiel. Ähnlich wie in unserer *Hallo-Welt*-Applikation wollen wir hier einen Namen einlesen und ihn ausgeben, allerdings nicht in der Methode `Main()` direkt, sondern in der Methode einer zweiten Klasse, von der wir eine Instanz erzeugen. Dieses Beispiel soll lediglich der Demonstration dienen und hat ansonsten keine Bedeutung. Im realen Leben würde vermutlich niemand so etwas programmieren.

```
/* Beispiel Variablendeklaration */
/* Autor:   Frank Eller          */
/* Sprache: C#                   */

using System;

class Ausgabe
{
  public void doAusgabe()
  {
    theValue = Console.ReadLine();
    Console.WriteLine("Hallo {0}",theValue);
  }

  public string theValue;
}
```

```
class Beispiel
{
  public static void Main()
  {
    Ausgabe myAusgabe = new Ausgabe();
    myAusgabe.doAusgabe();
  }
}
```

Sie finden das Programm auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_3\VARIABLEN.

Die Frage ist, ob dieses Beispiel wirklich funktioniert. Es wurde erklärt, dass die Deklaration von Methoden und Feldern gemischt werden kann, dass es also egal ist, wo genau ein Feld deklariert wird. In diesem Fall wird das Feld `theValue` nach der Methode `doAusgabe()` deklariert. Die Frage ist jetzt, ob diese Variable nicht zuerst hätte deklariert werden müssen.

Es funktioniert. Die Reihenfolge, in der die einzelnen Bestandteile der Klasse deklariert werden, ist wirklich egal, denn nach der Erzeugung der Instanz ist der Zugriff auf alle Felder der Klasse, die innerhalb des Gültigkeitsbereichs deklariert wurden, sichergestellt. Theoretisch könnten Sie Ihre Felder also auch zwischen den einzelnen Methoden deklarieren, für den Compiler macht das keinen Unterschied. Normalerweise ist es aber so, dass sich die Felddeklarationen entweder am Anfang oder am Ende der Klasse befinden, wiederum aus Gründen der Übersichtlichkeit.

Das reservierte Wort **void**, das in diesem Beispiel sowohl bei der Methode `Main()` als auch bei der Methode `doAusgabe()` verwendet wurde, haben wir auch schon kennen gelernt. Wie bereits in Kapitel 2 angesprochen, handelt es sich dabei um eine leere Rückgabe, d.h. die Methode liefert keinen Wert an die aufrufende Methode zurück. Sie werde **void** auch in Ihren eigenen Applikationen recht häufig verwenden, wenn Sie eine solche Methode schreiben, die lediglich einen Block von Anweisungen durchführt.

```
public void Ausgabe()
{
  Console.WriteLine("Hallo Welt");
}
```

Wenn Sie allerdings einen Wert zurückliefern, können Sie alle Standard-Datentypen von C# dafür verwenden. Eine Methode, die einen Wert zurückliefert, wird beim Aufruf behandelt wie eine Zuweisung, wobei auf den Datentyp geachtet werden muss. Die Variable, der der

Wert zugewiesen wird, muss exakt den gleichen Datentyp wie der gelieferte Wert haben, ansonsten funktioniert es nicht. C# achtet da peinlich genau darauf, es ist eine so genannte *typsichere* Sprache, bei der die Datentypen bei einer Zuweisung oder Parameterübergabe exakt übereinstimmen müssen.

***return***

Innerhalb der Methode wird ein Wert mittels der Anweisung **return** zurückgeliefert. Dabei handelt es sich um eine besondere Anweisung, die einerseits die Methode beendet (ganz gleich, ob noch weitere Anweisungen folgen) und sich andererseits auch wie eine Zuweisung verhält, da sie ja im Prinzip auch nichts anderes ist. Achten Sie immer darauf, dass der Datentyp des Werts, den Sie **return** zuweisen, mit dem Ergebnistyp übereinstimmt, da der Compiler ansonsten einen Fehler meldet.

```
/* Beispiel Ergebniswerte */
/* Autor:   Frank Eller   */
/* Sprache: C#           */

using System;

public class TestClass
{
  public int a;
  public int b;

  public int Addieren()
  {
    return a+b;
  }

public class MainClass
{
  public static void Main();
  {
    TestClass myTest = new TestClass();

    int myErgebnis;
    double ergebnis2;

    myTest.a = 10;
    myTest.b = 15;

    myErgebnis = myTest.Addieren();  //ok...
    ergebnis2  = myTest.Addieren();  //FEHLER!!
  }
}
```

Im Beispiel wird eine einfache Routine zum Addieren zweier Werte benutzt, um zu zeigen, dass C# tatsächlich auf die korrekte Übereinstimmung der verwendeten Datentypen achtet. Der Rückgabewert muss vom Datentyp her exakt mit dem Datentyp des Felds oder der Variable übereinstimmen, der er zugewiesen wird. Ist dies nicht der Fall, meldet der Compiler einen Fehler.

Die Zeile

```
myErgebnis = myTest.Addieren();
```

wird korrekt ausgeführt. `myErgebnis` ist als Variable mit dem Datentyp **`int`** deklariert, ebenso wie der Rückgabewert der Methode `Addieren()`. Keine Probleme hier. Anders sieht es bei der nächsten Zuweisung aus,

```
ergebnis2 = myTest.Addieren();
```

Da die Variable `ergebnis2` als **`double`** deklariert worden ist, funktioniert hier die Zuweisung nicht, der Compiler meldet einen Fehler.

Sie finden den Quelltext des Programms auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_3\ERGEBNISWERTE.

Der zurückzuliefernde Wert nach **`return`** wird oftmals auch in Klammern geschrieben, was nicht notwendig ist. Es ist jedoch vor allem für die Übersichtlichkeit innerhalb des Programmtextes sinnvoll, so dass ich im restlichen Buch ebenfalls so vorgehen werde. Auf die Geschwindigkeit des Programms zur Laufzeit hat es keinen Einfluss.

### 3.3.2 Variablen und Felder

Bisher haben wir nur die Deklaration von Feldern betrachtet. Wir wissen, dass Felder Daten aufnehmen und zur Verfügung stellen können und dass sie in einer Klasse deklariert werden. Es ist jedoch – wie wir im letzten Beispiel gesehen haben – auch möglich, Variablen innerhalb einer Methode zu deklarieren. Solche Variablen nennt man dann *lokale Variablen*.

***lokale Variablen***

Eine Variable, die innerhalb eines durch geschweifte Klammern bezeichneten Programmblocks deklariert wird, ist auch nur in diesem Block gültig. Sobald der Block verlassen wird, wird auch die Variable gelöscht. Diese Lokalität bezieht sich aber nicht nur auf die Anweisungsblöcke von Methoden, sondern, wie wir später auch noch in verschiedenen Beispielen sehen werden, auf jeden Anweisungsblock, den Sie programmieren. Anhand eines kleinen Beispielprogramms können Sie leicht kontrollieren, dass eine in einer Methode deklarierte Variable tatsächlich nur in dieser Methode gültig ist.

```
/* Beispiel lokale Variablen 1 */
/* Autor:   Frank Eller        */
/* Sprache: C#                 */


using System;

public class TestClass
{
  public static void Ausgabe()
  {
    Console.WriteLine("x hat den Wert {0}.",x);
  }

  public static void Main()
  {
    int x;
    x = Int32.Parse(Console.ReadLine());
    Ausgabe();
  }
}
```

Sie finden den Quelltext des Programms auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_3\LOKALE_VARIABLEN1.

Auf die verwendete Methode `Parse()` kommen wir im späteren Verlauf noch zu sprechen. Wenn Sie das kleine Programm eingeben und ausführen, werden Sie feststellen, dass der Compiler sich darüber beschwert, die Variable `x` in der Methode `Ausgabe()` nicht zu kennen. Damit hat er durchaus Recht, denn `x` ist lediglich in der Methode `Main()` deklariert und somit auch nur innerhalb dieser Methode gültig.

Was geschieht nun, wenn wir in der Methode `Ausgabe()` ebenfalls eine Variable `x` deklarieren? Nun, der Compiler wird die Variable klaglos annehmen und, falls sie initialisiert wurde, deren Wert ausgeben. Allerdings ist es unerheblich, welchen Wert wir unserer ersten Variable `x` zuweisen, denn diese ist nur innerhalb der Methode `Main()` gültig und hat somit keine Auswirkungen auf den Wert der Variable `x` in `Ausgabe()`. Ein kleines Beispiel macht dies deutlich:

```
/* Beispiel lokale Variablen 2 */
/* Autor:   Frank Eller        */
/* Sprache: C#                 */

using System;

public class TestClass
{
  public static void Ausgabe()
  {
    int x = 10;
    Console.WriteLine("x hat den Wert {0}.",x);
  }

  public static void Main()
  {
    int x;
    x = Int32.Parse(Console.ReadLine());
    Ausgabe();
  }
}
```

Ganz gleich, welchen Wert Sie auch eingeben, die Ausgabe des Programms wird immer lauten

**x hat den Wert 10.**

Sie finden das Pogramm auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_3\LOKALE_VARIABLEN2.

Eine Variable, die innerhalb eines Programmblocks deklariert wurde, ist nur für diesen Programmblock gültig. Sobald der Programmblock verlassen wird, wird die Variable und ihr Wert aus dem Speicher gelöscht. Dies gilt auch, wenn der Block innerhalb eines bestehenden Blocks deklariert ist. Jeder Deklarationsblock, der in geschweifte Klammern eingefasst ist, hat seinen eigenen lokalen Gültigkeitsbereich.

***Konstanten***

Es gibt aber noch eine Möglichkeit, Werte zu verwenden, nämlich die Konstanten. Sie werden durch das reservierte Wort `const` deklariert und verhalten sich eigentlich so wie Variablen, mit dem Unterschied, dass sie einerseits bei der Deklaration initialisiert werden *müssen* und andererseits ihr Wert nicht mehr geändert werden kann. Konstanten werden aber häufig nicht lokal innerhalb einer Methode verwendet, sondern eher als konstante Felder. In Kapitel 3.3.8 werden wir mehr über die Deklaration von Konstanten als Felder erfahren. Hier noch ein Beispiel für eine Konstante innerhalb einer Methode.

```
/* Beispiel Lokale Konstante */
/* Autor:   Frank Eller      */
/* Sprache: C#               /

using System;

public class Umfang
{
  public double Umfang(double d)
  {
    const double PI = 3,1415;
    return (d*PI);
  }
}

public class TestClass
{
  public static void Main()
  {
    double d;
    Umfang u = new Umfang();
    d = Double.Parse(Console.ReadLine());
    Console.WriteLine("Umfang: {0}",u.Umfang(d));
  }
}
```

Das Beispiel berechnet den Umfang eines Kreises, wobei der eingegebene Wert den Durchmesser darstellt.

Damit haben wir nun verschiedene Arten von Variablen kennen gelernt: einmal die Felder einer Klasse, die ja auch nur Variablen sind, weiterhin die statischen Felder einer Klasse, wobei es sich zwar um Variablen handelt, die sich aber von den herkömmlichen Feldern unterscheiden, und die lokalen Variablen, die nur jeweils innerhalb eines Programmblocks gültig sind. Diese drei Arten von Variablen tragen zur besseren Unterscheidung besondere Namen.

***Instanzvariablen***

Die herkömmlichen Felder einer Klasse, gleich ob sie `public` oder `private` deklariert sind, bezeichnet man auch als *Instanzvariablen*. Der Grund ist, dass sie erst verfügbar sind, wenn eine Instanz der entsprechenden Klasse erzeugt wurde.

***Klassenvariablen***

Statische Felder einer Klasse (die mit dem Modifikator `static` deklariert wurden) nennt man *statische Variablen* oder *Klassenvariablen*, da sie Bestandteil der Klassendefinition sind. Sie sind bereits verfügbar, wenn innerhalb des Programms der Zugriff auf die Klasse sichergestellt ist. Es muss keine Instanz der Klasse erzeugt werden.

**Lokale Variablen**

*Lokale Variablen* sind nur innerhalb des Programmblocks gültig, in dem sie deklariert wurden. Wird der Programmblock beendet, werden auch die darin deklarierten lokalen Variablen und ihre Werte gelöscht.

Sie werden feststellen, dass diese Begriffe auch im weiteren Verlauf des Buchs immer wieder auftauchen werden. Sie sollten sie sich deshalb gut einprägen.

### 3.3.3 this

Kommen wir zu einem ganz anderen Beispiel. Sehen Sie sich das folgende kleine Beispielprogramm an und versuchen Sie herauszufinden, welcher Wert ausgegeben wird.

```
/* Beispiel lokale Variablen 3 */
/* Autor:   Frank Eller       */
/* Sprache: C#               */

using System;

public class TestClass
{
  int x = 10;

  public void doAusgabe()
  {
    int x = 5;
    Console.WriteLine("x hat den Wert {0}.",x);
  }
}

public class Beispiel
{
  public static void Main()
  {
    TestClass tst = new TestClass();
    tst.doAusgabe();
  }
}
```

Na, worauf haben Sie getippt? Die Ausgabe lautet

**x hat den Wert 5.**

Innerhalb der Methode `doAusgabe()` wurde eine Variable `x` deklariert, wobei es sich um eine lokale Variable handelt. Auch wurde in der Klasse ein Feld mit Namen `x` deklariert, so dass man zu der Vermutung kommen könnte, es gäbe eine Namenskollision.

Für den Compiler jedoch sind beide Variablen in unterschiedlichen Gültigkeitsbereichen deklariert, wodurch es für ihn nicht zu einer Kollision kommen kann. Das Feld `x` ist Bestandteil der Klasse, die lokale Variable `x` Bestandteil der Methode `doAusgabe()`. Der Compiler nimmt sich, wenn nicht anders angegeben, die Variable, die er in der Hierarchie zuerst findet. Dabei sucht er zunächst innerhalb des Blocks, in dem er sich gerade befindet, und steigt dann in der Hierarchie nach oben. In diesem Fall ist die erste Variable, die er findet, die in der Methode `doAusgabe()` deklarierte lokale Variable `x`.

Sie finden das Programm auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_3\LOKALE_VARIABLEN3.

Wenn eine lokale Variable und ein Feld den gleichen Namen haben, muss es nicht zwangsläufig zu einer Kollision kommen. Der Compiler sucht vom aktuellen Standort aus nach einer Variablen oder einem Feld mit dem angegebenen Namen. Was zuerst gefunden wird, wird benutzt.

Es ist selbstverständlich auch möglich, innerhalb der Methode `doAusgabe()` auf das Feld x zuzugreifen, obwohl eine Variable mit diesem Namen existiert. Wir müssen dem Compiler nur mitteilen, dass er sich nicht um die lokale Variable x kümmern soll, sondern um das Feld, das in der Klasse deklariert ist. Dazu dient das reservierte Wort `this`.

***this***

`this` ist eine Referenz auf die aktuelle Instanz einer Klasse. Wenn eine Variable mittels `this` referenziert wird, wird auf das entsprechende Feld (falls vorhanden) der aktuellen Instanz der Klasse zugegriffen. Für unser Beispiel heißt das, wir müssen lediglich `x` mit `this.x` ersetzen:

```
/* Beispiel lokale Variablen 4 */
/* Autor:   Frank Eller        */
/* Sprache: C#                 */

using System;

public class TestClass
{
  int x = 10;

  public void doAusgabe()
  {
    int x = 5;
    Console.WriteLine("x hat den Wert {0}.",this.x);
  }
}

public class Beispiel
{
  public static void Main()
  {
    TestClass tst = new TestClass();
    tst.doAusgabe();
  }
}
```

Nun lautet die Ausgabe tatsächlich

**x hat den Wert 10.**

Sie finden das Programm auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_3\LOKALE_VARIABLEN4.

**this** ist eine Referenz auf die aktuelle Instanz einer Klasse. Das bedeutet, wird ein Variablenbezeichner mit this qualifiziert (z.B. this.x), so muss es sich um eine Instanzvariable handeln. Mit **this** kann nicht auf lokale Variablen zugegriffen werden.

Der Zugriff auf Felder bzw. Methoden der aktuellen Instanz einer Klasse mittels **this** ist natürlich ein sehr mächtiges Werkzeug. Wie umfangreich das Einsatzgebiet ist, lässt sich an dem kleinen Beispielprogramm natürlich nicht erkennen. Sie werden aber selbst des Öfteren in Situationen kommen, wo Sie bemerken, dass dieses kleine Wörtchen Ihnen eine große Menge Programmierarbeit sparen kann.

***Namenskollision***

Natürlich gibt es auch die Situation, dass der Compiler wirklich nicht mehr auseinander halten kann, welche Variable nun gemeint ist. In diesem Fall muss die Variable im innersten Block umbenannt wer-

den, um dem Compiler wieder eine eindeutige Unterscheidung zu ermöglichen.

Hierzu möchte ich ebenfalls ein Beispiel liefern, dazu muss ich aber auf eine Funktion zurückgreifen, die wir noch nicht kennen gelernt haben, nämlich eine Schleife. In diesem Beispiel soll es auch nur um die Tatsache gehen, dass es für die besagte Schleife ebenfalls einen Programmblock gibt, in dem wir natürlich auch lokale Variablen deklarieren können.

```
/* Beispiel lokale Variablen 5 */
/* Autor:   Frank Eller       */
/* Sprache: C#               */

using System;

public class TestClass
{
  int x = 10;

  public void doAusgabe()
  {
    bool check  = true;
    int myValue = 5;

    while (check)
    {
      int myValue = 10; //Fehler-myValue schon dekl.

      Console.WriteLine("Innerhalb der Schleife ...");
      Console.WriteLine("myValue: {0}",myValue);
      check = false;
    }
  }
}

public class Beispiel
{
  public static void Main()
  {
    TestClass tst = new TestClass();
    tst.doAusgabe();
  }
}
```

Sie finden das Programm auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_3\LOKALE_VARIABLEN5.

In diesem Beispiel deklarieren wir innerhalb der Methode `doAusgabe()` zunächst eine Variable `myValue`, die mit dem Initialwert 5 belegt wird. Das ist in Ordnung. Nun programmieren wir eine Schleife, in diesem Fall eine **`while`**-Schleife, die so lange wiederholt wird, bis der Wert der lokalen Variable `check` **`true`** wird. Die Schleife soll uns aber im Moment nicht interessieren, wichtig ist, dass sie einen eigenen Programmblock besitzt, in dem wir wieder eigene lokale Variablen deklarieren können.

Wir wissen, dass eine lokale Variable nur innerhalb des Blocks gültig ist, in dem ich sie deklariere. Im obigen Fall ist aber die zweite Deklaration von `myValue` nicht möglich, da es innerhalb der Methode bereits eine Variable mit diesem Namen gibt.

Der Grund hierfür ist, dass innerhalb einer Methode die Namen der lokalen Variablen untereinander eindeutig sein müssen. Ansonsten wäre es, um wieder auf das Beispiel zurückzukommen, nicht möglich innerhalb des Schleifenblocks auf die zuerst deklarierte Variable `myValue` zuzugreifen. Sie würde durch die zweite Deklaration verdeckt. Deshalb können Sie eine solche Deklaration nicht vornehmen.

Innerhalb einer Methode können Sie nicht zwei lokale Variablen mit dem gleichen Namen deklarieren, da eine der beiden verdeckt werden würde.

Ich werde versuchen, es auch noch auf eine andere Art verständlich zu machen. Nehmen wir an, wir hätten einen Programmblock deklariert. Dieser hat nun einen bestimmten Gültigkeitsbereich, in dem wir lokale Variablen deklarieren und Anweisungen verwenden können. Wenn wir nun innerhalb dieses Gültigkeitsbereichs einen weiteren Programmblock deklarieren, z.B. wie im Beispiel durch eine Schleife, dann ist dieser ja auch Bestandteil des bisherigen Gültigkeitsbereichs. Daher gelten auch die deklarierten Variablen für den neuen Block und dürfen nicht erneut deklariert werden.

### 3.3.4 Parameterübergabe

Methoden können Parameter übergeben werden, die sich dann innerhalb der Methode wie lokale Variablen verhalten. Deshalb funktioniert auch die Deklaration der Parameter wie bei herkömmlichen Variablen mittels Datentyp und Bezeichner, allerdings im Kopf der Methode.

Als Beispiel für die Parameterübergabe soll eine Methode dienen, die zwei ganzzahlige Werte auf ihre Größe hin kontrolliert. Ist der erste Wert größer als der zweite, wird **`true`** zurückgegeben, ansonsten **`false`**.

```
public bool isBigger(int a, int b)
{
  return (a>b);
}
```

Der Datentyp **bool**, der in diesem Beispiel verwendet wurde, steht für einen Wert, der nur zwei Zustände annehmen kann, nämlich wahr (**true**) oder falsch (**false**). Für das Beispiel gilt, dass der Wert, den der Vergleich `a>b` ergibt, zurückgeliefert wird. Ist `a` größer als `b`, wird **true** zurückgeliefert, denn der Vergleich ist wahr; ansonsten wird **false** zurückgeliefert.

Für die Parameter können natürlich keine Modifikatoren vergeben werden, das wäre ja auch unsinnig. Per Definitionem handelt es sich eigentlich um lokale Variablen (oder um eine Referenz auf eine Variable), so dass ohnehin nur innerhalb der Methode mit den Parametern gearbeitet werden kann.

### 3.3.5 Parameterarten

C# unterscheidet verschiedene Arten von Parametern. Die einfachste Art sind die Werteparameter, bei denen lediglich ein Wert übergeben wird, mit dem innerhalb der aufgerufenen Methode gearbeitet werden kann. Die beiden anderen Arten sind die Referenzparameter und die **out**-Parameter.

***Werteparameter***

Wenn Parameter auf die obige Art übergeben werden, nennt man sie *Werteparameter*. Die Methode selbst kann dann zwar einen Wert zurückliefern, die Werte der Parameter aber werden an die Methode übergeben und können in dieser verwendet werden, ohne die Werte in den ursprünglichen Variablen zu ändern. Intern werden in diesem Fall auch keine Variablen als Parameter übergeben, sondern nur deren Werte, auch dann, wenn Sie einen Variablenbezeichner angegeben haben. Die Parameter, die die Werte aufnehmen, gelten als lokale Variablen der Methode.

***Referenzparameter***

Was aber, wenn Sie einen Parameter nicht nur als Wert übergeben wollen, sondern als ganze Variable, d.h. der Methode ermöglichen wollen, die Variable selbst zu ändern? Auch hierfür gibt es eine Lösung, Sie müssen dann einen Referenzparameter übergeben.

***ref***

Referenzparameter werden durch das reservierte Wort **ref** deklariert. Es wird dann nicht nur der Wert übergeben, sondern eine *Referenz* auf die Variable, die den Wert enthält. Alle Änderungen, die an diesem Wert vorgenommen werden, werden auch an die ursprüngliche Variable weitergeleitet.

Wenn wir unser obiges Beispiel weiterverfolgen, könnte man statt des Rückgabewertes auch einen Referenzparameter übergeben, z.B. mit Namen `isOK`, und stattdessen den Rückgabewert weglassen.

```
public void IsBigger(int a, int b, ref bool isOK)
{
  isOK = (a>b);
}
```

Auf diese Art und Weise können Sie auch mehrere Werte zurückgeben, statt nur den Rückgabewert der Methode zu verwenden. Bei derartigen Methoden, die eine Aktion durchführen und dann eine größere Anzahl Werte mittels Referenzparametern zurückliefern, benutzt man als Rückgabewert auch gerne einen booleschen Wert, der den Erfolg der Operation anzeigt. Der eigentliche Datentransfer geschieht dann über die Referenzparameter.

***ref beim Aufruf***

Wenn Sie eine Methode mit Referenzparameter aufrufen, dürfen Sie nicht vergessen, das reservierte Wort **ref** auch beim Aufruf zu verwenden. Der Grund dafür liegt wie so oft darin, dass C# absolut typsicher ist und keinerlei Kompromisse eingeht. Wenn Sie **ref** beim Aufruf nicht angeben, geht der Compiler davon aus, dass Sie einen Wert übergeben wollen. Er erwartet aber aufgrund der Methodendeklaration eine Referenz auf die Variable, also liefert er Ihnen eine Fehlermeldung.

Werteparameter übergeben Werte. Referenzparameter übergeben eine Referenz auf eine Variable. Da dies ein Unterschied ist, muss bei Referenzparametern sowohl in der Methode, die aufgerufen wird, als auch beim Aufruf das reservierte Wort **ref** verwendet werden.

Außerdem ist es möglich, bei Werteparametern wirklich nur mit Werten zu arbeiten und diese direkt zu übergeben. Bei Referenzparametern kann das nicht funktionieren, da diese ja einen Verweis auf eine Variable erwarten, damit sie deren Wert ändern können.

Als letztes Beispiel für Parameter hier noch eine Methode, die zwei Zahlenwerte vertauscht. Diese Methode arbeitet nur mit Referenzparametern und liefert keinen Wert zurück.

```
public void Swap(ref int a, ref int b)
{
  int c = a;
  a = b;
  b = c;
}
```

**out-Parameter**

Für Parameter, die mit dem reservierten Wort **out** deklariert werden, gilt im Prinzip das Gleiche wie für die **ref**-Parameter. Es sind ebenfalls Referenzparameter, mit den gleichen Eigenarten. Auch hier wird die Änderung an der Variable an die Variable in der aufrufenden Methode weitergeleitet, ebenso müssen Sie beim Aufruf das Schlüsselwort **out** mit angeben.

Sie werden sich nun fragen, warum hier zwei verschiedene Schlüsselwörter benutzt werden können, wenn doch das Gleiche gemeint ist. Nun, es gibt einen großen Unterschied, der allerdings normalerweise nicht auffällt. Wir wissen bereits, dass Variablen vor ihrer ersten Verwendung initialisiert werden müssen. Die Übergabe einer Variablen als Parameter gilt als erste Verwendung, folglich müssen wir ihr vorher einen Wert zuweisen. Das gilt auch für **ref**-Parameter, nicht aber für Parameter, die mit **out** übergeben werden.

**out**

Wenn Sie eine Variable mit **out** übergeben, muss diese nicht vorher initialisiert werden. Das kann auch in der aufgerufenen Methode geschehen, wichtig ist nur, dass es dort auch geschehen muss, bevor die Variable auf irgendeine Art verwendet wird. Das folgende Beispiel zeigt, wie das funktioniert. Wir werden wieder die kurze Routine zur Überprüfung auf größer oder kleiner verwenden, diesmal übergeben wir den Parameter `isOk` aber als **out**-Parameter und initialisieren ihn vor dem Methodenaufruf nicht.

```
/* Beispiel out-Parameter */
/* Sprache: C#            */
/* Autor:   Frank Eller   */

using System;

class TestClass
{
  public static void IsBigger(int a, int b,out bool isOk)
  {
    isOk = (a>b);  //Erste Zuweisung=Initialisierung
  }

  public static void Main()
  {
    bool isOk;  //nicht initialisiert ...
    int a;
    int b;

    a = Console.ReadLine().ToInt32();
    b = Console.ReadLine().ToInt32();
```

```
    IsBigger(a,b,out isOk);

    Console.WriteLine("Ergebnis a>b: {0}",isOk);
  }
}
```

Die Variable `isOk` wird erst in der Methode `IsBigger()` initialisiert, in unserem Fall sogar direkt mit dem Ergebniswert. Sie finden das Programm auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_3\OUT-PARAMETER.

Parameter, die mit **ref** oder **out** übergeben werden, unterscheiden sich nur dadurch, dass ein **ref**-Parameter vor der Übergabe initialisiert sein muss, ein **out**-Parameter nicht.

### 3.3.6 Überladen von Methoden

Das Überladen von Methoden ist eine sehr nützliche und zeitsparende Sache. Es handelt sich dabei um die Möglichkeit, mehrere Methoden mit dem gleichen Namen zu deklarieren, die aber unterschiedliche Funktionen durchführen. Unterscheiden müssen sie sich anhand der Übergabeparameter, der Compiler muss sich die Methode also eindeutig heraussuchen können.

Ein gutes Beispiel hierfür ist eine Rechenfunktion, bei der Sie mehrere Werte zueinander addieren. Wäre es nicht möglich, Methoden zu überladen, müssten Sie für jede Addition eine eigene Methode mit einem eigenen Namen deklarieren, sich diesen Namen merken und später im Programm auch noch immer die richtige Methode aufrufen. Durch die Möglichkeit des Überladens können Sie sich auf einen Namen beschränken und mehrere gleich lautende Methoden mit unterschiedlicher Parameteranzahl zur Verfügung stellen. Eingebettet in eine eigene Klasse sieht das dann so aus:

```
/* Beispiel Methoden überladen 1 */
/* Autor:   Frank Eller          */
/* Sprache: C#                   */

using System;

public class Addition
{
  public int Addiere(int a, int b)
  {
    return a+b;
  }
```

```
  public int Addiere(int a, int b, int c)
  {
    return a+b+c;
  }

  public int Addiere(int a, int b, int c, int d)
  {
    return a+b+c+d;
  }
}

public class Beispiel
{
  public static void Main()
  {
    Addition myAdd = new Addition();

    int a = Console.ReadLine().ToInt32();
    int b = Console.ReadLine().ToInt32();
    int c = Console.ReadLine().ToInt32();
    int d = Console.ReadLine().ToInt32();

    Console.WriteLine("a+b     = {0}",myAdd.Addiere(a,b));
    Console.WriteLine("a+b+c   = {0}",myAdd.Addiere(a,b,c));
    Console.WriteLine("a+b+c+d = {0}",
                          myAdd.Addiere(a,b,c,d));
  }
}
```

Sie finden das Programm auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_3\ÜBERLADEN1.

Wenn diese Klasse bzw. diese Methoden innerhalb Ihres Programms benutzt werden, genügt ein Aufruf der Methode `Addiere()` mit der entsprechenden Anzahl Parameter. Der Compiler sucht sich die richtige Methode heraus und führt sie aus.

Die obige Klasse kann auch noch anders geschrieben werden. Sehen Sie sich die folgende Klasse an und vergleichen Sie sie dann mit der oberen:

```
/* Beispiel Methoden überladen 2 */
/* Autor:   Frank Eller          */
/* Sprache: C#                   */

using System;

public class Addition
```

```
{
  public int Addiere(int a, int b)
  {
    return a+b;
  }

  public int Addiere(int a, int b, int c)
  {
    return Addiere(Addiere(a,b),c);
  }

  public int Addiere(int a, int b, int c, int d)
  {
    return Addiere(Addiere(a,b,c),d);
  }
}
```

Sie finden ein entsprechendes Programm mit dieser Klasse ebenfalls auf der beiliegenden CD, im Verzeichnis BEISPIELE\KAPITEL_3\ÜBERLADEN2.

Im obigen Beispiel werden einfach die bereits bestehenden Methoden verwendet. Auch das ist möglich, da C# sich automatisch die passende Methode heraussucht. Denken sie aber immer daran, dass es sehr schnell passieren kann, bei derartigen Methodenaufrufen in eine unerwünschte Rekursion zu gelangen (z.B. wenn Sie einen Parameter zu viel angeben und die Methode sich dann selbst aufrufen will ... nun, irgendwann wird es auch in diesem Fall einen Fehler geben ☺).

Beim Überladen der Methoden müssen Sie darauf achten, dass diese sich in den Parametern unterscheiden. Der Compiler muss die Möglichkeit haben, die verschiedenen Methoden eindeutig zu unterscheiden, was über die Anzahl bzw. Art der Parameter geschieht.

Der Ergebniswert der Methode hat dabei keinen Einfluss. Die Deklaration zweier Methoden mit gleichem Namen, gleichen Parametern, aber unterschiedlichem Ergebniswert ist nicht möglich.

In C# sind viele bereits vorhandene Methoden ebenso in diversen unterschiedlichen Versionen vorhanden. So haben Sie sicherlich schon bemerkt, dass unsere häufig verwendete Methode `WriteLine()` mit den verschiedensten Parameterarten umgehen kann. Einmal übergeben wir lediglich einen Wert, dann eine Zeichenkette oder auch eine Zeichenkette mit Platzhaltern. Auch hier handelt es sich eigentlich nur um eine einfache überladene Methode, bei der der Compiler sich die richtige heraussucht.

### 3.3.7 Statische Methoden/Variablen

Die statischen Methoden haben wir bereits kennen gelernt, und wir wissen mittlerweile, dass wir für deren Verwendung keine Instanz der Klasse erzeugen müssen. Man sagt auch, die Attribute einer Klasse, die als `static` deklariert sind, sind ein Bestandteil der Klasse selbst; die anderen Attribute sind nach der Instanziierung Bestandteile des jeweiligen Objekts.

Das bedeutet auch Folgendes: Wenn mehrere Instanzen einer Klasse erzeugt wurden und in jeder dieser Instanzen wird eine statische Methode aufgerufen, dann ist das immer dieselbe Methode – sie ist nämlich Bestandteil der Klassendefinition selbst und nicht des Objekts. In Abbildung 3.3 wird im Bild dargestellt, wie sich das Ganze verhält.

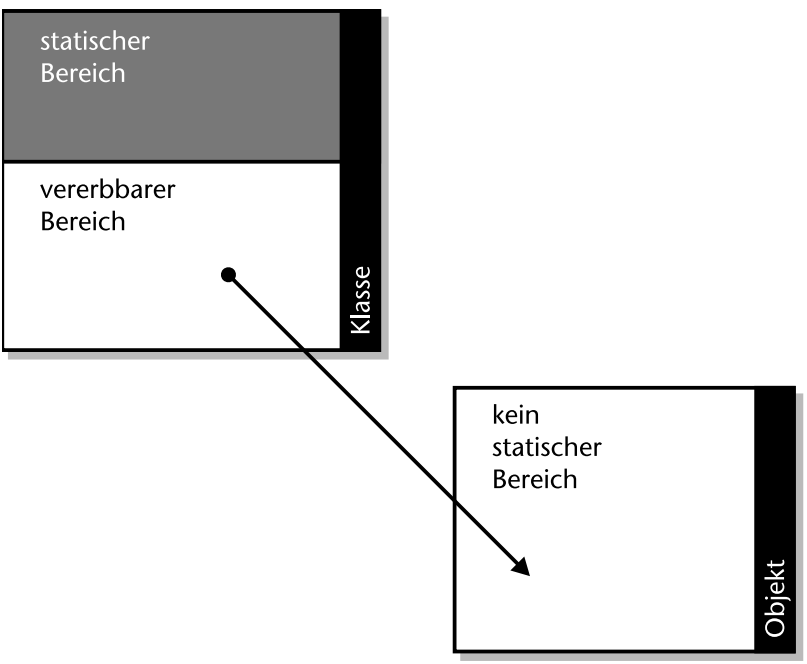


*Abbildung 3.3: Statischer und vererbbarer Bereich*

***globale Variablen***

Überlegen wir doch einmal, wie es dann mit den Variablen bzw. Feldern der Klasse aussieht. Wenn eine Variable als `static` deklariert ist, also Bestandteil der Klasse selbst und nicht des aus der Klasse erzeugten Objekts ist, dann müsste diese Variable praktisch global gültig sein – über alle Instanzen hinweg.

Exakt so ist es. Ein Beispiel soll uns das verdeutlichen. Für dieses Beispiel wird ein Fahrzeugverleih angenommen, der sowohl Fahrräder als auch Motorräder als auch Autos verleiht. Der Besitzer will nun immer wissen, wie viele Autos, Fahrräder und Motorräder unterwegs sind. Wir erstellen also eine entsprechende Klasse `Fahrzeug`, aus der wir dann die entsprechenden benötigten Objekte erstellen können:

```
/* Beispielklasse statische Felder 1 */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

public class Fahrzeug
{
  int anzVerliehen;

  public void Ausleihen()
  {
    anzVerliehen++;
  }

  public void Zurueck()
  {
    anzVerliehen--;
  }

  public int GetAnzahl()
  {
    return anzVerliehen;
  }
}
```

Die Variable `anzVerliehen` zählt unsere verliehenen Fahrzeuge. Mit den beiden Methoden `Ausleihen()` und `Zurueck()`, die beide als **public** deklariert sind, können Fahrzeuge verliehen werden. Die Methode `GetAnzahl()` schließlich liefert die Anzahl verliehener Fahrzeuge zurück, denn die Variable `anzVerliehen` ist ja als **private** deklariert (Sie erinnern sich: ohne Modifikator wird in Klassen als Standard die Sichtbarkeitsstufe **private** verwendet).

Damit funktioniert unsere Klasse bereits, wenn wir eine Instanz davon erstellen. Doch nun will der Fahrzeugverleih auch automatisch eine Übersicht aller verliehenen Fahrzeuge bekommen. Nichts leichter als das. Wir fügen einfach eine statische Variable hinzu und schon haben wir einen Zähler, der alle verliehenen Fahrzeuge unabhängig vom Typ zählt.

```
/* Beispielklasse statische Felder 2 */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

public class Fahrzeug
{
  int anzVerliehen;
  static int anzGesamt = 0;

  public void Ausleihen()
  {
    anzVerliehen++;
    anzGesamt++;
  }

  public void Zurueck()
  {
    anzVerliehen--;
    anzGesamt--;
  }

  public int GetAnzahl()
  {
    return anzVerliehen;
  }
}
```

Als Letztes wollen wir nun noch eine Methode hinzufügen, mit der wir erfahren können, wie viele Fahrzeuge insgesamt verliehen sind. Wenn wir die statische Variable `anzGesamt` nämlich veröffentlichen würden, könnte sie innerhalb des Programms geändert werden. Das soll aber nicht erlaubt sein. Also belassen wir es bei der Sichtbarkeitsstufe **private** und fügen lieber noch eine Methode hinzu, die wir ausnahmsweise ebenfalls statisch machen.

```
/* Beispielklasse statische Felder 3 */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

public class Fahrzeug
{
  int anzVerliehen;
  static int anzGesamt = 0;
```

```
  public void Ausleihen()
  {
    anzVerliehen++;
    anzGesamt++;
  }

  public void Zurueck()
  {
    anzVerliehen--;
    anzGesamt--;
  }

  public int GetAnzahl()
  {
    return anzVerliehen;
  }

  public static int GetGesamt();
  {
    return anzGesamt;
  }
}
```

Innerhalb einer statischen Methode können Sie nur auf lokale und statische Variablen zugreifen. Die anderen Variablen sind erst verfügbar, wenn eine Instanz der Klasse erzeugt wurde, und da das nicht Voraussetzung für den Aufruf einer statischen Methode ist, können Sie die Instanzvariablen auch nicht verwenden.

In C# ist es nicht möglich, „richtige“ globale Variablen zu deklarieren, weil alle Deklarationen innerhalb einer Klasse vorgenommen werden müssen. Ohne Klassen geht es hier nun mal nicht. Durch das Konzept der statischen Variablen haben Sie aber die Möglichkeit, dennoch allgemeingültige Variablen zu erzeugen.

Deklarieren Sie einfach eine Klasse mit Namen `Glb` oder `Global`, in der Sie alle globalen Variablen zusammenfassen. Deklarieren Sie diese als statische Felder und ermöglichen Sie es allen Programmteilen, auf die Klasse zuzugreifen.

### 3.3.8 Deklaration von Konstanten

Oft kommt es vor, dass man festgelegte Werte mehrfach innerhalb eines Programms verwenden möchte. Beispiele hierfür gibt es viele, in der Elektrotechnik z.B. die Werte 1.4142 bzw. 1.7320, oder auch den Umrechnungskurs für den Euro, der in Buchhaltungsprogrammen wichtig ist. Natürlich ist es recht mühselig, diese Werte immer wieder komplett eintippen zu müssen. Stattdessen können Sie die Werte fest in einer Klasse ablegen und immer wieder mit Hilfe des entsprechenden Bezeichners darauf zugreifen.

Das Problem ist, dass diese Werte verändert werden könnten. Um sie unveränderlich zu machen, könne sie sie auch als so genannte Konstanten festlegen, indem Sie das reservierte Wort **const** benutzen.

```
/* Beispiel Konstanten 1 */
/* Autor:   Frank Eller  */
/* Sprache: C#          */

using System;

public class glb
{
  public const double Wurzel2 = 1,4142;
  public const double Wurzel3 = 1,7320;
}
```

Sie haben bereits den Modifikator **static** kennen gelernt. Dementsprechend werden Sie jetzt vermuten, dass bei der obigen Deklaration vor der Verwendung der Werte eine Instanz der Klasse `glb` erzeugt werden muss.

Dem ist nicht so. Alle Konstanten, die im obigen Beispiel deklariert wurden, sind statisch. Der Modifikator **static** ist in Konstantendeklarationen nicht erlaubt. Dass Konstanten immer statisch sind, ist durchaus logisch, denn sie haben ohnehin immer den gleichen Wert.

Wenn Sie Felder einer Klasse als Konstanten deklarieren, sind diese immer statisch. Der Modifikator **static** darf nicht im Zusammenhang mit Konstantendeklarationen verwendet werden.

Der Zugriff auf die oben deklarierten Konstanten funktioniert daher wie folgt:

```
/* Beispiel Konstanten 2 */
/* Autor:   Frank Eller  */
/* Sprache: C#          */

using System;

public class glb
{
  public const double W2 = 1,4142;
  public const double W3 = 1,7320;
}

public class TestClass
{
  public static void Main()
  {
    //Ausgabe der Konstanten
    //der Klasse glb

    Console.WriteLine("Wurzel 2: {1}\nWurzel 3: {2}",glb.W2,glb.W3);
  }
}
```

### 3.3.9 Zugriff auf statische Methoden/Variablen

Statische Methoden und Variablen sind wie bereits gesagt Bestandteil der Klasse selbst. Das bedeutet, dass auf sie anders zugegriffen werden muss als auf Instanzmethoden bzw. -variablen. Sehen wir uns die folgende Deklaration einmal an:

```
/* Beispielklasse statische Methoden */
/* Autor:   Frank Eller              */
/* Sprache: C#                       */


public class TestClass
{

  public int myValue;

  public static bool SCompare(int theValue)
  {
    return (theValue>0);
  }
```

```
  public bool Compare(int theValue)
  {
    return (myValue==theValue);
  }
}
```

Die Methode `SCompare()` ist eine statische Methode, die Methode `Compare()` eine Instanzmethode, die erst nach der Erzeugung einer Instanz verfügbar ist. Wenn wir nun auf die Methode `SCompare()` zugreifen wollen, können wir dies nicht über das erzeugte Objekt tun, stattdessen müssen wir den Bezeichner der Klasse verwenden, denn die statische Methode ist kein Bestandteil des Objekts – sie ist ein Bestandteil der Klasse, aus der wir das Objekt erstellt haben.

```
/* Beispiel statische Felder (Main) */
/* Autor:   Frank Eller             */
/* Sprache: C#                      */

using System;

public class Beispiel
{
  public static void Main()
  {
    TestClass myTest = new TestClass();

    //Kontrolle mittels SCompare
    bool Test1 = TestClass.SCompare(5);

    //Kontrolle mittels Compare
    myTest.myValue = 0;
    bool Test2 = myTest.Compare(5);
  }
}
```

Das komplette Programm (incl. `Main()`-Methode und der Klasse `TestClass`) finden Sie wie üblich auf der beiliegenden CD, im Verzeichnis BEISPIELE\KAPITEL_3\STATISCHE_METHODEN.

Auch hier soll wieder eine Abbildung zeigen, wie sich der Aufruf von statischen Methoden von dem der Instanzmethoden unterscheidet. In Abbildung 3.4 sehen Sie den Unterschied zwischen den verschiedenen Arten des Aufrufs.

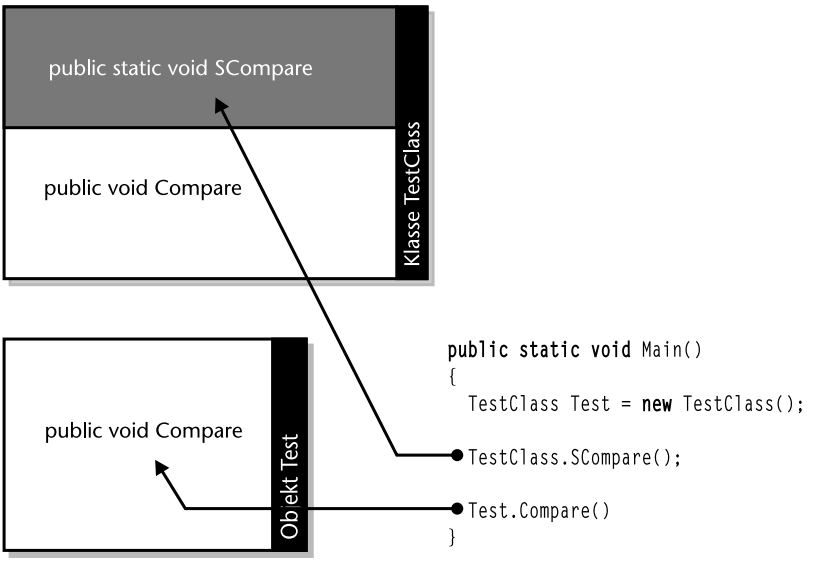


*Abbildung 3.4: Aufruf von statischer und Instanzmethode*

Bei statischen Methoden wie auch bei statischen Variablen muss zur Qualifizierung der Bezeichner der Klasse selbst benutzt werden, da statische Elemente Bestandteil der Klasse und nicht des erzeugten Objekts sind.

Für Objekte gilt, dass über sie nur auf Instanzmethoden bzw. Instanzvariablen zugegriffen werden kann.

### 3.3.10 Konstruktoren und Destruktoren

Beim Erzeugen eines Objekts aus einer Klasse mit dem Operator `new` wird der so genannte *Konstruktor* einer Klasse aufgerufen. Dabei handelt es sich um eine besondere Methode, die dazu dient, Variablen zu initialisieren und für jedes neue Objekt einen Ursprungszustand herzustellen. Das Gegenstück dazu ist der *Destruktor*, der aufgerufen wird, wenn das Objekt wieder aus dem Speicher entfernt wird. Um diesen werden wir uns aber an dieser Stelle nicht kümmern, denn die Garbage-Collection nimmt uns die Arbeit mit dem Destruktor komplett ab. Kümmern wir uns also um die Initialisierung unseres Objekts.

***Der Konstruktor***

Der Konstruktor ist eine Methode ohne Rückgabewert (auch ohne `void` – es wird kein Datentyp angegeben) und mit dem Modifikator `public`, damit man von außen darauf zugreifen kann. Der Name des

Konstruktors entspricht dem Namen der Klasse. Für unsere Fahrzeugklasse würde eine solche Deklaration also folgendermaßen aussehen:

```
public Fahrzeug()
{
  //Anweisungen zur Initialisierung
}
```

Eine Klasse muss dabei nicht zwingend nur einen Konstruktor zur Verfügung stellen (den Standard-Konstruktor stellt sie automatisch zur Verfügung). Der Programmierer kann auch mehrere Konstruktoren erstellen, die sich durch ihre Parameter unterscheiden. Damit ist es möglich, z.B. einen Standard-Konstruktor ohne übergebene Parameter zu erstellen, der dann das Objekt mit 0 verliehenen Fahrzeugen erzeugt, und einen weiteren, bei dem man die Anzahl der verliehenen Fahrzeuge angibt.

```
/* Beispielklasse statische Felder + Konstruktor */
/* Autor:   Frank Eller                          */
/* Sprache: C#                                   */

public class Fahrzeug
{
  int anzVerliehen;
  static int anzGesamt = 0;

  public Fahrzeug()  // Der Standard-Konstruktor
  {
    anzVerliehen = 0;
  }

  public Fahrzeug(int Verliehene)  //Der zweite Konstruktor
  {
    anzVerliehen = Verliehene;
    anzGesamt += Verliehene;
  }

  public void Ausleihen()
  {
    anzVerliehen++;
    anzGesamt++;
  }

  public void Zurueck()
  {
    anzVerliehen--;
    anzGesamt--;
  }
```

```
  public int GetAnzahl()
  {
    return anzVerliehen;
  }

  public static int GetGesamt();
  {
    return anzGesamt;
  }
}
```

Der Operator +=, der im obigen Beispiel auftaucht, bedeutet, dass der rechts stehende Wert dem Wert in der links stehenden Variable hinzuaddiert wird. Passend dazu gibt es dann auch den -=-Operator, der eine Subtraktion bewirkt. Anders ausgedrückt: `x += y` entspricht `x = x+y` und `x -= y` entspricht `x = x-y`. Diese Operatoren nennt man zusammengesetzte Operatoren, da sie eine Berechnung und eine Zuweisung zusammenfassen.

***Destruktor***

Wenn wir über Konstruktoren sprechen, müssen wir auch das Gegenteil ansprechen, nämlich den Destruktor. Aber eigentlich ist es in C# kein richtiger Destruktor, er dient eher der Finalisierung, d.h. den Aufräumarbeiten, wenn die Instanz der Klasse aus dem Speicher entfernt wird.

In anderen Programmiersprachen ist es teilweise so, dass der Destruktor explizit aufgerufen werden muss, um den Speicher, der für die Instanz der Klasse reserviert wurde, freizugeben. Das erledigt aber in C# die Garbage Collection, die ja automatisch arbeitet. Sie können jedoch einen Destruktor für eigene Aufräumarbeiten deklarieren. Allerdings wissen Sie nie, wann er aufgerufen wird, weil das von der Garbage Collection zu einem passenden Zeitpunkt erledigt wird.

***Destruktor deklarieren***

Ein Destruktor wird ebenso deklariert wie ein Konstruktor, allerdings mit einer Tilde vor dem Bezeichner. Die Tilde (~) ist das Zeichen für das Einerkomplement in C#, auf deutsch: die Bedeutung wird umgedreht.

Wenn unsere Klasse aus dem Speicher entfernt wird, sollte es eigentlich der Fall sein, dass die Anzahl der verliehenen Fahrzeuge dieser Fahrzeugart von der Anzahl der insgesamt verliehenen Fahrzeuge abgezogen wird. Dazu können wir den Destruktor verwenden.

```
/* Beispielklasse statische Felder + Destruktor */
/* Autor:   Frank Eller                         */
/* Sprache: C#                                  */

public class Fahrzeug
{
  int anzVerliehen;
  static int anzGesamt = 0;

  public Fahrzeug()
  {
    anzVerliehen = 0;
  }

  public Fahrzeug(int Verliehene)
  {
    anzVerliehen = Verliehene;
    anzGesamt += Verliehene;
  }

  public ~Fahrzeug()  //Der Destruktor
  {
    anzGesamt -= anzVerliehen;
  }

  public void Ausleihen()
  {
    anzVerliehen++;
    anzGesamt++;
  }

  public void Zurueck()
  {
    anzVerliehen--;
    anzGesamt--;
  }

  public int GetAnzahl()
  {
    return anzVerliehen;
  }

  public static int GetGesamt();
  {
    return anzGesamt;
  }
}
```

Ein Destruktor hat keine Parameter und auch keinen Rückgabewert. Vor allem aber: Er wird vom Compiler automatisch aufgerufen, Sie müssen sich nicht darum kümmern. Auch hinkt das obige Beispiel ein wenig, denn es würde ja voraussetzen, dass wir den Destruktor selbst aufrufen, was wir natürlich nicht tun. Nehmen Sie dieses Beispiel einfach als Anschauungsobjekt. Normalerweise werden Sie nie einen Destruktor deklarieren müssen, es wird auch davon abgeraten, weil sich das Laufzeitverhalten des Programms zum Schlechten ändern kann.

Damit hätten wir alles, was im Moment über Klassen zu sagen wäre, abgehandelt. Mit diesen Informationen sind Sie eigentlich schon in der Lage, kleinere Programme zu schreiben. Was noch fehlt, sind die Möglichkeiten wie Schleifen, Verzweigungen usw., die wir ja noch nicht besprochen haben, ohne die aber eine sinnvolle Programmierung nicht möglich ist. Bevor wir jedoch dazu kommen, zunächst noch eine weitere Möglichkeit der Programmunterteilung, nämlich die Namensräume.

## 3.4 Namensräume

Klassen sind nicht die einzige Möglichkeit, ein Programm in verschiedene Bereiche aufzuteilen. Ein Konzept, das auch schon in C++ oder Java Verwendung fand und ebenso in C# enthalten ist, sind die so genannten *Namensräume*, oder im Original *namespaces*. Diese Art der Unterteilung wurde auch innerhalb des .net-Frameworks verwendet, und um genau zu sein, haben wir Namensräume schon von Anfang an verwendet. Denn alle Datentypen, die wir bisher in unseren Beispielen (auch schon im *Hallo-Welt*-Programm) verwendet haben, sind im Namensraum `System` deklariert.

***Namensräume***

Ein Namensraum bezeichnet einen Gültigkeitsbereich für Klassen. Innerhalb eines Namensraums können mehrere Klassen oder sogar weitere Namensräume deklariert werden. Dabei ist ein Namensraum nichts zwangsläufig auf eine Datei beschränkt, innerhalb einer Datei können mehrere Namensräume deklariert werden, ebenso ist es möglich, einen Namensraum über zwei oder mehrere Dateien hinweg zu deklarieren.

### 3.4.1 Namensräume deklarieren

***namespace***

Die Deklaration eines Namensraums geschieht über das reservierte Wort `namespace`. Darauf folgen geschweifte Klammern, die den Gültigkeitsbereich des Namensraums angeben, also eben so, wie die ge-

schweiften Klammern bei einer Methode den Gültigkeitsbereich für lokale Variablen angeben. Der Unterschied ist, dass in einem Namensraum nur Klassen deklariert werden können, aber keine Methoden oder Variablen – die gehören dann in den Gültigkeitsbereich der Klasse.

Die Deklaration eines Namensraums mit der Bezeichnung *CSharp* würde also wie folgt aussehen:

```
namespace CSharp
{

  //Hier die Deklarationen innerhalb des Namensraums

}
```

**Dateiunabhängigkeit**

Wenn die Datei für weitere Klassen nicht ausreicht oder zu unübersichtlich werden würde, kann in einer weiteren Datei der gleiche Namensraum deklariert werden. Beide Dateien zusammen wirken dann wie ein Namensraum, d.h. der Gültigkeitsbereich ist der gleiche – alle Klassen, die in einer der Dateien deklariert sind, können unter dem gleichen Namensraum angesprochen werden.

Seien Sie vorsichtig, wenn Sie Namensräume in mehreren Dateien verteilen. Zwar ist es problemlos möglich, es kann aber zu Konflikten kommen, wenn Sie verschiedene Bestandteile einer Klasse in unterschiedlichen Dateien, aber im gleichen Namensraum deklarieren. Eine Klasse und alle Bestandteile, die darin verwendet werden, sollten immer in einer Datei deklariert sein.

### 3.4.2 Namensräume verschachteln

Namensräume können auch verschachtelt werden. Die Bezeichner des übergeordneten und des untergeordneten Namensraums werden dann wie gewohnt mit einem Punkt getrennt. Wenn wir einen Namensraum mit der Bezeichnung `CSharp.Lernen` deklarieren möchten, also `CSharp` als übergeordneten und `Lernen` als untergeordneten Namensraum, haben wir zwei Möglichkeiten. Wir können beide Namensräume getrennt deklarieren, einen innerhalb des Gültigkeitsbereichs des anderen, wodurch die Möglichkeit gegeben ist, für jeden Namensraum getrennt Klassen zu deklarieren:

```
namespace CSharp
{

  //Hier die Deklarationen für CSharp

  namespace Lernen
  {

    //Hier die Deklarationen für CSharp.Lernen

  }

}
```

Die zweite Möglichkeit ist die, den Namensraum `CSharp.Lernen` direkt zu deklarieren, wieder mit Hilfe des Punkts:

```
namespace CSharp.Lernen
{

  //Hier die Deklarationen für CSharp.Lernen

}
```

Diese Möglichkeit empfiehlt sich dann, wenn ein Namensraum thematisch einem anderen untergeordnet sein soll, Sie aber die beiden dennoch in getrennten Dateien unterbringen wollen. Da die Deklaration eines Namensraums nicht in der gleichen Datei vorgenommen werden muss, ist es also egal, wo ich den Namensraum deklariere.

### 3.4.3 Verwenden von Namensräumen

Wenn eine Klasse verwendet werden soll, die innerhalb eines Namensraums deklariert ist, gibt es zwei Möglichkeiten. Die erste Möglichkeit besteht darin, den Bezeichner des Namensraums vor den Bezeichner der Klasse zu schreiben und beide mit einem Punkt zu trennen:

```
CSharp.SomeClass.SomeMethod();
```

***using***

Die zweite Möglichkeit ist die, den gesamten Namensraum einzubinden, wodurch der Zugriff auf alle darin enthaltenen Klassen ohne explizite Angabe des Namensraum-Bezeichners möglich ist. Dies wird bewirkt durch das reservierte Wort `using`. Normalerweise wird ein Namensraum am Anfang eines Programms bzw. einer Datei eingebunden:

```
using CSharp;
using CSharp.Lernen;

SomeClass.SomeMethod();
```

Der wohl am häufigsten benutzte Namensraum, der in jedem Programm eigentlich auch benötigt wird, ist der Namensraum `System`. Am Anfang Ihrer Programme sollte dieser also immer eingebunden werden. In einigen der diversen Beispiele des Buchs haben Sie das schon gesehen, auch beim ersten Programm *Hallo Welt* sind wir bereits so vorgegangen.

Namensräume sind sehr effektiv und es wird intensiv Gebrauch davon gemacht. Vor allem, weil alle Standardroutinen in Namensräumen organisiert sind, habe ich diese Informationen unter Basiswissen eingeordnet. Sie werden sich sicherlich schnell daran gewöhnen, Namensräume zu verwenden und auch selbst für Ihre eigenen Applikationen zu deklarieren.

### 3.4.4 Der globale Namensraum

Das Einbinden bereits vorhandener Namensräume ist eine Sache, das Erstellen eigener Namensräume eine andere. Sie dürfen selbstverständlich für jede Klasse oder für jedes Programm einen Namensraum deklarieren, Sie sind aber nicht dazu gezwungen. Wenn Sie bei der Programmierung direkt mit der ersten Klasse beginnen, ohne einen Namensraum zu deklarieren, wird das Programm genauso laufen.

*globaler Namensraum*

In diesem Fall sind alle Klassen im so genannten globalen Namensraum deklariert. Dieser ist stets vorhanden und dementsprechend müssen Sie keinen eigenen deklarieren. Es ist jedoch sinnvoller, vor allem bei größeren Applikationen, doch von dieser Möglichkeit Gebrauch zu machen.

## 3.5 Zusammenfassung

In diesem Kapitel haben Sie die Basis der Programmierung mit C# kennen gelernt, nämlich die Klassen und die Namensräume. Außerdem haben wir uns ein wenig mit den Modifikatoren befasst, die Sie bei Deklarationen immer wieder benötigen.

Klassen und Namensräume dienen der Strukturierung eines Programms in einzelne Teilbereiche. Dabei stellen Namensräume so etwas wie einen Überbegriff dar (z.B. könnte ein Namensraum CSharpLernen als Namensraum für alle Klassen des Buchs dienen, und für

die einzelnen Kapitel könnte dieser Namensraum weiter unterteilt werden), Klassen stellen die Funktionalität her und auch eine Möglichkeit zum Ablegen der Daten, wobei es sich wieder um eine Untergliederung handelt.

Ein Beispiel für eine sinnvolle Unterteilung sind die verschiedenen Datentypen von C#, einige haben wir ebenfalls in diesem Kapitel angesprochen. Während alle im gleichen Namensraum deklariert sind, handelt es sich doch um unterschiedliche Klassen, d.h. die Datentypen sind wiederum unterteilt. Mit den Klassen, die Sie in Ihren Programmen verwenden, wird es genauso sein – für bestimmte Aufgaben erstellen Sie jeweils eine Klasse, innerhalb des Programms arbeiten diese Klassen zusammen und stellen so die Gesamtfunktionalität her.

## 3.6 Kontrollfragen

Auch in diesem Kapitel wieder einige Fragen, die den Inhalt ein wenig vertiefen sollen.

1. Von welcher Basisklasse sind alle Klassen in C# abgeleitet?
2. Welche Bedeutung hat das Schlüsselwort `new`?
3. Warum sollten Bezeichner für Variablen und Methoden immer eindeutige, sinnvolle Namen tragen?
4. Welche Sichtbarkeit hat das Feld einer Klasse, wenn kein Modifikator bei der Deklaration benutzt wurde?
5. Wozu dient der Datentyp `void`?
6. Was ist der Unterschied zwischen Referenzparametern und Werteparametern?
7. Welche Werte kann eine Variable des Typs `bool` annehmen?
8. Worauf muss beim Überladen einer Methode geachtet werden?
9. Innerhalb welchen Gültigkeitsbereichs ist eine lokale Variable gültig?
10. Wie kann eine globale Variable deklariert werden, ohne das Konzept der objektorientierten Programmierung zu verletzen?
11. Wie kann ich innerhalb einer Methode auf ein Feld einer Klasse zugreifen, selbst wenn eine lokale Variable existiert, die den gleichen Bezeichner trägt wie das Feld, auf das ich zugreifen will?
12. Wie kann ich einen Namensraum verwenden?
13. Mit welchem reservierten Wort wird ein Namensraum deklariert?
14. Für welchen Datentyp ist `int` ein Alias?
15. In welchem Namensraum sind die Standard-Datentypen von C# deklariert?

## 3.7 Übungen

Für die Übungen gilt: Schreiben Sie für jede Übung auch eine Methode `Main()`, mit der Sie die Funktion überprüfen können. Es handelt sich hierbei nicht um komplizierte Arbeiten, es geht lediglich darum, sicherzustellen, dass die Klasse funktioniert.

### Übung 1

Deklarieren Sie eine Klasse, in der Sie einen String-Wert, einen Integer-Wert und einen Double-Wert speichern können.

### Übung 2

Erstellen Sie für jedes der drei Felder einen Konstruktor, so dass das entsprechende Feld bereits bei der Instanziierung mit einem Wert belegt werden kann.

### Übung 3

Erstellen Sie eine Methode, in der zwei Integer-Werte miteinander multipliziert werden. Es soll sich dabei um eine statische Methode handeln.

### Übung 4

Erstellen Sie drei Methoden um den Feldern Werte zuweisen zu können. Der Name der drei Methoden soll gleich sein.

### Übung 5

Erstellen Sie eine Methode, mit der einem als Parameter übergebenen String der in der Klasse als Feld gespeicherte String hinzugefügt werden kann. Um zwei Strings aneinander zu fügen, können Sie den +-Operator benutzen, Sie können sie also ganz einfach addieren. Die Methode soll keinen Wert zurückliefern.

# 4 Datenverwaltung

Programme tun eigentlich nichts anderes, als Daten zu verwalten und damit zu arbeiten. Auf der einen Seite haben wir die bereits besprochenen Methoden, in denen wir Anweisungen zusammenfassen können, die etwas mit unseren Daten tun. Auf der anderen Seite stehen die Daten selbst. In diesem Kapitel wollen wir uns nun mit den grundlegenden Datentypen beschäftigen und aufzeigen, wie man damit arbeitet.

## 4.1 Datentypen

### 4.1.1 Speicherverwaltung

***Arten von Datentypen***

C# kennt zwei Sorten von Datentypen, nämlich einmal die wertebehafteten Typen, kurz auch *Wertetypen* genannt, und dann die *Referenztypen*. Der Unterschied besteht in der Art, wie die Werte gespeichert werden. Während bei Wertetypen der eigentliche Wert direkt gespeichert wird, speichert ein Referenztyp lediglich einen Verweis. Wertetypen werden in C# grundsätzlich auf dem so genannten *Stack* gespeichert, Referenztypen auf dem so genannten *Heap*.

***Stack***

Als Programmierer müssen Sie sich des Öfteren mit solchen Ausdrücken wie *Stack* und *Heap* herumschlagen, aus diesem Grund hier auch die Erklärung, auch wenn sie eigentlich erst bei wirklich komplexen Programmierproblemen eine Rolle spielen. Als Stack bezeichnet man einen Speicherbereich, in dem Daten einfach abgelegt werden, solange sie gebraucht werden, z.B. bei lokalen Variablen oder Methodenparametern. Jedes mal, wenn eine neue Methode aufgerufen oder eine Variable deklariert wird, wird eine Kopie der Daten erzeugt.

Freigegeben werden die Daten in dem Moment, in dem sie nicht mehr benötigt werden. Das bedeutet, in dem Moment, in dem Sie

eine Variable deklarieren, wird Speicher reserviert in der Größe, die dem maximalen Wert entspricht, den die Variable enthalten kann. Dieser wird natürlich festgelegt über die Art des Datentyps, z.B. 32 Bit (4 Byte) beim Datentyp `int`.

**Heap**

Mit dem Heap sieht es ganz anders aus. Speicher auf dem Heap muss angefordert werden und kann, wenn er nicht mehr benötigt wird, auch wieder freigegeben werden. Das darin enthaltene Objekt wird dabei gelöscht. Wenn Sie Instanzen von Klassen erzeugen, wird der dafür benötigte Speicher beim Betriebssystem angefordert und dieses kümmert sich darum, dass Ihr Programm den Speicher auch bekommt.

Programmiersprachen wie z.B. C++ erforderten, dass der angeforderte Speicher explizit wieder freigegeben wird, d.h es wird darauf gewartet, dass Sie selbst im Programm die entsprechende Anweisung dazu geben. Geschieht dies nicht, kommt es zu so genannten *Speicherleichen*, d.h. Speicher ist und bleibt reserviert, obwohl das Programm, das ihn angefordert hat, längst nicht mehr läuft. Ein weiteres Manko ist, dass bei einem Neustart des Programms vorher angeforderter Speicher nicht mehr erkannt wird – wenn Sie also vergessen, Speicher freizugeben, wird irgendwann Ihr Betriebssystem die Meldung „Speicher voll“ anzeigen und eine weitere Zusammenarbeit verweigern.

### 4.1.2 Die Null-Referenz

Es ist möglich, dass ein Objekt zwar erzeugt ist, aber keinen Inhalt besitzt. Das beste Beispiel hierfür sind *Delegates*, auf die wir später noch zu sprechen kommen werden. Kurz gesagt sind Delegates eine Möglichkeit, auf verschiedene Funktionen zuzugreifen, die alle die Form des Delegate haben (also gleichen Rückgabewert und gleiche Parameter, aber unterschiedliche Funktion). Es ist jedoch nicht zwingend notwendig, dass ein Delegate auf eine Methode verweist.

**null**

Auch Delegates sind natürlich grundsätzlich Objekte und natürlich ist es auch möglich – wie in angesprochenem Beispiel – dass ein Objekt einfach auf nichts verweist. Sie können das mit dem fest definierten Wert `null` kontrollieren, dem Standardwert für Objekte, die noch keine Referenz besitzen. `null` ist der Standardwert für alle Referenztypen.

Der Wert `null` ist der Standardwert für alle Referenztypen. Er ist eine Referenz, die auf kein Objekt (also sozusagen ins „Leere“) verweist.

### 4.1.3 Garbage-Collection

Mit C# hat die Angst vor Speicherleichen ein Ende, was auch bereits in der Einführung angesprochen wurde. Das .net-Framework, also die Basis für C# als Programmiersprache, bietet eine automatische *Garbage-Collection*, die nicht benötigten Speicher auf dem Heap automatisch freigibt. Der Name bedeutet ungefähr so viel wie „Müllabfuhr“, und genau das ist auch die Funktionsweise – der „Speichermüll“ wird abtransportiert.

*new*

In C# werden Sie deshalb kaum einen Unterschied zwischen Wertetypen und Referenztypen feststellen, außer dem, dass Referenztypen stets mit dem reservierten Wort **new** erzeugt werden müssen, während bei Wertetypen in der Regel eine einfache Zuweisung genügt.

*object*

Hinzu kommt, dass alle Datentypen in C# wirklich von einer einzigen Klasse abstammen, nämlich der Klasse **object**. Das bedeutet, dass Sie nicht nur Methoden in anderen Klassen implementiert haben können, die mit den Daten arbeiten, vielmehr besitzen die verschiedenen Datentypen selbst bereits einige Methoden, die grundlegende Funktionalität bereitstellen. Dabei ist es vollkommen egal, ob es sich um Werte- oder Referenztypen handelt.

### 4.1.4 Methoden von Datentypen

Wie wir in Kapitel 3 bereits gesehen haben, gibt es zwei verschiedene Arten von Methoden, nämlich einmal die *Instanzmethoden*, die nur für die jeweilige Instanz des Datentyps gelten, und dann die *Klassenmethoden* oder statischen Methoden, die Bestandteil der Klasse selbst sind und sich nicht um die erzeugte Instanz scheren.

*Parse*

So ist die Methode `Parse()` z.B. eine Klassenmethode. Wenn Sie nun eine Variable des Datentyps **int** deklariert haben, können Sie die Methode `Parse()` dazu verwenden, den Inhalt eines Strings in den Datentyp **int** umzuwandeln. Diese Methode ist in allen numerischen Datentypen implementiert, falls Sie also einen 32-Bit-Integer-Wert verwenden wollen, sähe die Deklaration folgendermaßen aus:

```
int i = Int32.Parse(myString);
```

**Aliase**

In diesem Fall muss für den Aufruf von `Parse()` der Datentyp `Int32` verwendet werden, der eigentliche Datentyp. Das reservierte Wort `int` steht hier lediglich für einen Alias, d.h. es hat die gleiche Bedeutung, ist aber nicht die ursprüngliche Bezeichnung des Datentyps. Bei der Verwendung einer statischen Methode muss zur Qualifikation die wirkliche Basisklasse verwendet werden, und das ist im Falle von `int` die Klasse `Int32`.

Alternativ könnten Sie auch die Instanzmethode (Achtung, hier kommt schon ein Unterschied) des umzuwandelnden String benutzen, um daraus einen Integer-Wert zu erzeugen:

```
int i = myString.ToInt32();
```

Der Unterschied zwischen beiden Methoden ist auf den ersten Blick nicht ersichtlich, tun doch beide im Prinzip das Gleiche. `Parse()` ist allerdings eine überladene Methode, d.h. sie existiert in mehreren Varianten und sie berücksichtigt auch die landesspezifischen Einstellungen des Betriebssystems. Daher ist sie im Allgemeinen vorzuziehen. Mehr zu den Umwandlungsmethoden und zu `Parse()` noch in Kapitel 4.2.5.

**Typsicherheit**

C# ist eine typsichere Sprache, und somit sind die Datentypen auch nicht frei untereinander austauschbar. In C++ konnte man z.B. die Datentypen `int` und `bool` sozusagen zusammen verwenden, denn jeder Integer-Wert größer als 0 lieferte den booleschen Wert `true` zurück. In C# ist dies nicht mehr möglich. Hier ist jeder Datentyp autonom, d.h. einem booleschen Wert kann kein ganzzahliger Wert zugewiesen werden. Stattdessen müssten Sie, wollten Sie das gleiche Resultat erzielen, eine Kontrolle durchführen, die dann einen booleschen Wert zurückliefert. Wir werden im weiteren Verlauf dieses Kapitels noch ein Beispiel dazu sehen.

**Wert- und Typumwandlung**

Dennoch kann ein Datentyp auf mehrere Arten in einen anderen Datentyp umgewandelt werden. Eine Möglichkeit, z.B. aus einem String, der eine Zahl enthält, einen Integer-Wert zu machen, haben wir bereits kennen gelernt. Allerdings ist es aufgrund der Typsicherheit auch so, dass sogar zwei numerische Datentypen nicht einander zugeordnet werden können, wenn z.B. der Quelldatentyp einen größeren Wertebereich als der Zieldatentyp besitzt. Hier muss eine explizite Konvertierung stattfinden, ein so genanntes *Casting*, wodurch C# gezwungen wird, die Datentypen zu konvertieren. Dabei handelt es sich also um eine Wertumwandlung, während das obige Beispiel eine Typumwandlung darstellt.

Damit genug zur Einführung. Kümmern wir uns nun um die Standard-Datentypen von C#.

### 4.1.5 Standard-Datentypen

Einige Datentypen haben wir schon kennen gelernt, darunter der Datentyp `int` für die ganzen Zahlen und der Datentyp `double` für die reellen Zahlen. Alle diese Datentypen sind unter dem Namensraum `System` deklariert, den Sie in jedes Ihrer Programme mittels `using` einbinden sollten. Tabelle 4.1 gibt Ihnen nun einen Überblick über die Standard-Datentypen von C#.

| Alias | Größe | Bereich | Datentyp |
|---|---|---|---|
| sbyte | 8 Bit | -128 bis 127 | SByte |
| byte | 8 Bit | 0 bis 255 | Byte |
| char | 16 Bit | Nimmt ein 16-Bit Unicode-Zeichen auf | Char |
| short | 16 Bit | -32768 bis 32767 | Int16 |
| ushort | 16 Bit | 0 bis 65535 | UInt16 |
| int | 32 Bit | -2147483648 bis 2147483647. | Int32 |
| uint | 32 Bit | 0 bis 4294967295 | UInt32 |
| long | 64 Bit | –9223372036854775808 bis 9223372036854775807 | Int64 |
| ulong | 64 Bit | 0 bis 18446744073709551615 | UInt64 |
| float | 32 Bit | $\pm 1.5 \times 10^{-45}$ bis $\pm 3.4 \times 10^{38}$ (auf 7 Stellen genau) | Single |
| double | 64 Bit | $\pm 5.0 \times 10^{-324}$ bis $\pm 1.7 \times 10^{308}$ (auf 15-16 Stellen genau) | Double |
| decimal | 128 Bit | $1.0 \times 10^{-28}$ bis $7.9 \times 10^{28}$ (auf 28-29 Stellen genau) | Decimal |
| bool | 1 Bit | `true` oder `false` | Boolean |
| string | unb. | Nur begrenzt durch Speicherplatz, für Unicode-Zeichenketten | String |

*Tabelle 4.1: Die Standard-Datentypen von C#*

Die Standard-Datentypen bilden die Basis, es gibt aber noch weitere Datentypen, die Sie verwenden bzw. selbst deklarieren können. Doch dazu später mehr. An der Tabelle können Sie sehen, dass die hier angegebenen Datentypen eigentlich nur Aliase sind, die eigentlichen Datentypen des .net-Frameworks sind im Namensraum System unter den in der letzten Spalte angegebenen Namen deklariert. So ist z.B. `int` ein Alias für den Datentyp `System.Int32`.

***Arten von Datentypen***

In der obigen Tabelle finden sich drei Arten von Datentypen, nämlich einmal die ganzzahligen Typen (auch *Integrale Typen* genannt), dann die Gleitkommatypen und die Datentypen `string`, `bool` und `char`. `string` und `char` dienen der Aufnahme von Zeichen (`char`)

bzw. Zeichenketten (`string`), alle im Unicode-Format. Das bedeutet, jedes Zeichen belegt 2 Byte, somit können pro verwendetem Zeichensatz 65535 verschiedene Zeichen dargestellt werden. Die ersten 255 Zeichen ensprechen dabei der ASCII-Tabelle, die Sie auch im Anhang des Buchs finden. Der Datentyp `bool` entspricht einem Ja/Nein-Typ, d.h. er hat genau zwei Zustände, nämlich `true` und `false`.

Alle Standard-Datentypen der obigen Tabelle bis auf den Datentyp `string` sind Wertetypen. Da Strings nur durch die Größe des Hauptspeichers begrenzt sind, kann es sich nicht um Wertetypen handeln, denn diese haben eine festgelegte Größe. Strings hingegen sind dynamisch, d.h. hierbei handelt es sich um einen Referenztyp.

### 4.1.6 Type und typeof()

C# ist, wie bereits öfters angesprochen, eine typsichere Sprache. Zu den Eigenschaften einer solchen Sprache gehört auch, dass man immer ermitteln kann, welchen Datentyp eine Variable hat, oder sogar, von welcher Klasse sie abgeleitet ist. All das ist in C# problemlos möglich. Während für die Konvertierung bereits Methoden von den einzelnen Datentypen selbst implementiert werden, stellt C# für die Arbeit mit den Datentypen selbst die Klasse `Type` zur Verfügung, die im Namensraum `System` deklariert ist. Außerdem kommt der Operator `typeof` zum Einsatz, wenn es darum geht, herauszufinden, welchen Datentyp ein Objekt oder eine Variable besitzt.

***typeof***

Der Operator `typeof` wird eigentlich eingesetzt wie eine Methode, denn das Objekt, dessen Datentyp ermittelt werden soll, wird ihm in Klammern übergeben. Es kann sich allerdings nicht um eine Methode handeln, denn wie wir wissen, ist eine Methode immer Bestandteil einer Klasse. Ein Methodenaufruf wird immer durch die Angabe entweder des Klassennamens (bei statischen Methoden) oder des Objektnamens (bei Instanzmethoden) qualifiziert. Daran, dass dies hier nicht der Fall ist, können wir erkennen, dass es sich bei `typeof` um einen Bestandteil der Sprache selbst handeln muss.

***Type***

Der Rückgabewert, den `typeof` liefert, ist vom Datentyp `Type`. Dieser repräsentiert eine Typdeklaration, d.h. mit `Type` lässt sich mehr über den Datentyp eines Objekts bzw. einer Variablen herausfinden. Und auch wenn es nicht so aussieht, es gibt vieles, was man über eine Variable erfahren kann. Unter anderem liefert `Type` Methoden zur Bestimmung des übergeordneten Datentyps, zur Bestimmung der Attribute eines Datentyps oder zum Vergleich zweier Datentypen. Die folgende Liste gibt Ihnen einen Überblick über die Methoden des Datentyps `Type`.

### Instanzmethoden von Type

```
public override bool Equals(object o)
public new bool Equals(Type t)
```

Die Methode `Equals()` kontrolliert, ob der Datentyp, den die aktuelle Instanz von Type repräsentiert, dem Datentyp des übergebenen Parameters entspricht. Der Rückgabewert ist ein boolescher Wert, der Parameter ist entweder vom Typ **object** oder `Type`.

```
public FieldInfo GetField(string name)
public abstract FieldInfo GetField(
    BindingFlags bindingAttr
);
```

Die Methode `GetField()` liefert Informationen zu dem Feld mit dem angegebenen Namen zurück. Der zurückgelieferte Wert ist vom Typ `FieldInfo`, der wiederum von der Klasse `MemberInfo` abgeleitet ist.

```
public FieldInfo[] GetFields()
public abstract FieldInfo[] GetFields(
    string name;
    BindingFlags bindingAttr
);
```

Die Methode `GetFields()` liefert ein Array des Typs `FieldInfo` zurück, in dem alle Felder des Datentyps aufgelistet sind. BindingFlags ist eine Klasse, mit der Bedingungen übergeben werden können. Nur wenn ein Feld diesen Bedingungen genügt, wird es auch zurückgeliefert bzw. in das Ergebnisarray aufgenommen.

```
public MemberInfo[] GetMember(string name)
public virtual MemberInfo[] GetMember(
    string name;
    BindingFlags bindingAttr
);
public virtual MemberInfo[] GetMember(
    string name;
    MemberTypes type;
    BindingFlags bindingAttr
);
```

Die Methode `GetMember()` liefert Informationen über das spezifizierte Attribut der Klasse zurück. Der Ergebniswert ist ein Array, da es mehrere Attribute mit gleichem Namen geben kann (z.B. bei überladenen Methoden). Mit dem Parameter `bindingAttr` können wieder Bedingun-

gen übergeben werden, der Parameter `type` vom Typ `MemberTypes` enthält eine genauere Spezifikation für den Typ des Attributs. Damit sind Sie in der Lage festzulegen, welche Arten von Attributen (nur Methoden, nur Eigenschaften) zurückgeliefert werden dürfen.

```
public MemberInfo[] GetMembers();
public abstract MemberInfo[] GetMembers(
    BindingFlags bindingAttr
);
```

Die Methode `GetMembers()` liefert ebenfalls ein Array des Typs `MemberInfo` zurück, bezieht sich jedoch auf alle Attribute der Klasse. Auch hier können Sie über einen Parameter vom Typ `BindingFlags` wieder Ansprüche an die zurückgelieferten Attribute setzen.

```
public MethodInfo GetMethod(string name);
public MethodInfo GetMethod(
    string name;
    Type[] types
);
public MethodInfo GetMethod(
    string name;
    BindingFlags bindingAttr
);
public MethodInfo GetMethod(
    string name;
    Type[] types;
    ParameterModifier[] modifiers
);
public MethodInfo GetMethod(
    string name;
    BindingFlags bindingAttr;
    Binder binder;
    Type[] types;
    ParameterModifier[] modifiers
);
public MethodInfo GetMethod(
    string name;
    BindingFlags bindingAttr;
    Binder binder;
    CallingConventions callConvention;
    Type[] types;
    ParameterModifier[] modifiers
);
```

Die vielfach überladene Methode GetMethod() liefert Informationen über eine bestimmte, durch die Parameter spezifizierte Methode zurück. Den Typ BindingFlags kennen wir schon aus den anderen Methoden. Neu ist der Parameter modifiers vom Typ ParameterModifier, mit dem angegeben werden kann, welche Modifikatortypen für die Methode gelten müssen, damit sie zurückgeliefert wird. Der Parameter binder vom Typ Binder steht für Eigenschaften, die die zurückgelieferte Methode haben muss. callConvention vom Typ CallingConventions steht für Aufrufkonventionen der Methode. Dazu gehört z.B. die Art, wie die Parameter übergeben werden, wie der Ergebniswert zurückgeliefert wird usw. Sie können mit den verschiedenen Parametern genau festlegen, welche Methoden zurückgeliefert werden sollen und welche nicht.

```
public MethodInfo[] GetMethods();
public abstract MethodInfo[] GetMethods(
    BindingFlags bindingAttr
);
```

Die Methode GetMethods() liefert alle öffentlichen Methoden des Datentyps zurück, ggf. genauer spezifiziert durch den Parameter bindingAttr.

```
public PropertyInfo GetProperty(string name);
public PropertyInfo GetProperty(
    string name;
    BindingFlags bindingAttr
);
public PropertyInfo GetProperty(
    string name;
    Type[] types;
);
public PropertyInfo GetProperty(
    string name;
    Type[] types;
    ParameterModifier[] modifiers
);
public PropertyInfo GetProperty(
    string name;
    BindingFlags bindingAttr;
    Binder binder;
    Type[] types;
    ParameterModifier[] modifiers
);
```

Die Methode GetProperty() liefert analog zu den anderen besprochenen Get-Methoden die durch die Parameter spezifizierten Eigen-

schaften des Datentyps zurück. Die Klasse PropertyInfo ist ebenfalls von der Klasse MemberInfo abgeleitet.

```
public PropertyInfo[] GetProperties();
public abstract PropertyInfo[] GetProperties(
    BindingFlags bindingAttr
);
```

Die Methode GetProperties() liefert alle öffentlichen Eigenschaften des Datentyps, ggf. genauer spezifiziert durch den Parameter bindingAttr, zurück.

```
public virtual bool IsInstanceOfType(object o);
```

Die Methode IsInstanceOfType() kontrolliert, ob der aktuelle Datentyp eine Instanz der angegebenen Klasse bzw. des angegebenen Datentyps ist.

```
public virtual bool IsSubClassOf(Type o);
```

Die Methode IsSubClassOf() kontrolliert, ob der aktuelle Datentyp eine Unterklasse der angegebenen Klasse bzw. des angegebenen Datentyps ist. D. h. es wird kontrolliert, ob der aktuelle Datentyp vom angegebenen Datentyp abgeleitet ist.

### Type verwenden

Eine der häufigsten Verwendungen von Type wird vermutlich die Kontrolle des Datentyps sein. Das folgende Beispiel zeigt, wie man Type benutzen kann, um zwei Datentypen miteinander zu vergleichen.

```
/* Beispiel Typkontrolle */
/* Autor:   Frank Eller  */
/* Sprache: C#          */

using System;

class TestClass
{
  public static void Main()
  {
    int x = 200;

    Type t = typeof(Int32);
```

```
    if (t.Equals(x.GetType()))
      Console.WriteLine("x ist vom Typ Int32.");
    else
      Console.WriteLine("x ist nicht vom Typ Int32.");
  }
}
```

*GetType()*

Die Methode `GetType()`, die jeder Datentyp zur Verfügung stellt, liefert dabei ebenfalls einen Datentyp zurück, nämlich den des aktuellen Objekts. Im Beispiel wird der Datentyp `Int32` mit dem Datentyp der Variable `x` verglichen. Da diese vom Typ `int` ist, einem Alias für den Datentyp `Int32`, sollte der Vergleich eigentlich positiv ausfallen. Wenn Sie das Programm ausführen, werden Sie feststellen, dass dem tatsächlich so ist. Die Ausgabe lautet:

**x ist vom Typ Int32.**

Das obige Beispiel finden Sie auch auf der CD, im Verzeichnis BEISPIELE\KAPITEL_4\TYPKONTROLLE.

### Eigenschaften von Type

Im Beispiel haben wir kontrolliert, ob der Datentyp der Variable `x` dem Datentyp `Int32` entspricht. Gleichzeitig haben wir den Nachweis erbracht, dass `int` lediglich ein Alias für `Int32` ist, dass es sich also um den gleichen Datentyp handelt. Die Kontrolle muss allerdings nicht zwingend auf diese Art durchgeführt werden, wenn Sie z.B. nur erfahren wollen, ob es sich bei dem angegebenen Typ um einen Wertetyp, eine Aufzählung oder einen Referenztyp handelt. Hierfür bietet der Datentyp `Type` auch einige Eigenschaften, die nur zum Lesen sind und genauere Informationen über den Datentyp liefern. Es handelt sich dabei zumeist um boolesche Eigenschaften. Tabelle 4.2 listet einige oft verwendete Eigenschaften auf.

| Eigenschaft | Bedeutung |
|---|---|
| BaseType | Liefert den Namen des Datentyps zurück, von dem der aktuelle Datentyp direkt abgeleitet ist. |
| FullName | Liefert den vollen Namen des Datentyps incl. des Namens des Namensraums, in dem er deklariert ist, zurück. |
| IsAbstract | Liefert `true` zurück, wenn es sich um einen abstrakten Type handelt, von dem abgeleitet werden muss. Abstrakte Klassen behandeln wir in Kapitel 8.1.4. |
| IsArray | Liefert `true` zurück, wenn es sich bei dem angegebenen Datentyp um ein Array handelt. |

*Tabelle 4.2: Eigenschaften von Type*

| Eigenschaft | Bedeutung |
|---|---|
| IsByRef | Liefert **true** zurück, wenn es sich bei dem Datentyp um eine Referenz auf eine Variable handelt (Parameter, die als **ref**- oder **out**-Parameter übergeben wurden). |
| IsClass | Liefert **true** zurück, wenn es sich bei dem Datentyp um eine Klasse handelt. |
| IsEnum | Liefert **true** zurück, wenn es sich bei dem Datentyp um einen Aufzählungstyp handelt. Aufzählungstypen, so genannte Enums, werden wir in Kapitel 7.3 behandeln. |
| IsInterface | Liefert **true** zurück, wenn es sich bei dem Datentyp um ein Interface handelt. Informationen über Interfaces finden Sie in Kapitel 8.2. |
| IsNotPublic | Liefert **true** zurück, wenn der Datentyp nicht als öffentlich deklariert ist. |
| IsPublic | Liefert **true** zurück, wenn der Datentyp als öffentlich deklariert ist. |
| IsSealed | Liefert **true** zurück, wenn der Datentyp versiegelt ist, d.h. nicht von ihm abgeleitet werden kann. Versiegelte Klassen werden wir in Kapitel 8.1.5 behandeln. |
| IsValueType | Liefert **true** zurück, wenn es sich bei dem Typ um einen Wertetyp handelt. |
| Namespace | Liefert den Bezeichner des Namensraums zurück, in dem der Typ deklariert ist. |

*Tabelle 4.2: Eigenschaften von Type (Forts.)*

Mit Hilfe dieser Eigenschaften können Sie sehr schnell mehr über einen bestimmten Datentyp erfahren.

Der Datentyp `Type` repräsentiert eine Typdeklaration. Er dient dazu, mehr über einen Datentyp herauszufinden. Wenn Sie den Datentyp einer Variable herausfinden wollen, können Sie die Instanzmethode `GetType()` verwenden; falls Ihnen der Datentyp bekannt ist, können Sie auch den Operator **typeof** verwenden. Beide liefern einen Wert vom Typ `Type` zurück.

## 4.2 Konvertierung

Wir haben die Typsicherheit von C# bereits angesprochen, auch die Tatsache, dass es nicht wie z.B. in C++ möglich ist, einen Integer-Wert einer booleschen Variable zuzuweisen. Um dies zu verdeutlichen möchte ich nun genau dieses Beispiel - also den Vergleich zwischen C# und C++ – darstellen.

In C++ ist die folgende Zuweisung durchaus möglich:

```
/* Beispiel Typumwandlung (C++) */

/* Autor:   Frank Eller         */
/* Sprache: C++                 */

void Test()
{
  int testVariable = 100;
  bool btest;

  btest = testVariable;
}
```

Der Wert der booleschen Variable `btest` wäre in diesem Fall **true**, weil in C++ jeder Wert größer als 0 als **true** angesehen wird. 0 ist der einzige Wert, der **false** ergibt.

In C# ist die obige Zuweisung nicht möglich. Die Datentypen **int** und **bool** unterscheiden sich in C#, daher ist eine direkte Zuweisung nicht möglich. Man müsste den Code ein wenig abändern und den booleschen Wert mittels einer Abfrage ermitteln. Dazu benutzen wir den Operator != für die Abfrage auf Ungleichheit:

```
/* Beispiel Typumwandlung (C#)  */

/* Autor:   Frank Eller         */
/* Sprache: C#                  */

void Test
{
  int testVariable = 100;
  bool btest;

  btest = (testVariable != 0);
}
```

In diesem Fall wird der booleschen Variable `btest` dann der Wert **true** zugewiesen, wenn die Variable `testVariable` nicht den Wert 0 hat. In C# ist diese Art der Zuweisung für einen solchen Fall unbedingt notwendig.

### 4.2.1 Implizite Konvertierung

Manchmal ist es jedoch notwendig, innerhalb eines Programms Werte von einem Datentyp in den anderen umzuwandeln. Hierfür

gibt es in C# die *implizite* und die *explizite* Konvertierung. Außerdem stellen die Datentypen auch noch Methoden für die Konvertierung zur Verfügung. Aber der Reihe nach.

***implizite Konvertierung***

Wenn Sie einen Zahlenwert in einer Variable vom Typ `short` abgelegt haben, wissen Sie, dass dieser Datentyp einen gewissen Bereich beinhaltet, in dem sich der Zahlenwert befinden darf. Es ist ebenso klar, dass der Datentyp `int`, der ja einen größeren Wertebereich besitzt, ebenfalls verwendet werden könnte. Damit wird folgende Zuweisung möglich:

```
int i;
short s = 100;

i = s;
```

`i` hat den größeren Wertebereich, der in `s` gespeicherte Wert kann daher einfach aufgenommen werden. Dabei wird der Datentyp des Werts konvertiert, d.h. aus dem `short`-Wert wird automatisch ein `int`-Wert. Eine solche Konvertierung, die wir eigentlich nicht als solche wahrnehmen, bezeichnet man als *implizite Konvertierung*. Es wird dabei zwar tatsächlich eine Konvertierung vorgenommen, allerdings fällt uns das nicht auf. Den Grund dafür liefert Ihnen auf anschaulichere Weise Abbildung 4.1, die klarmacht, warum Sie nichts von der Konvertierung mitbekommen.

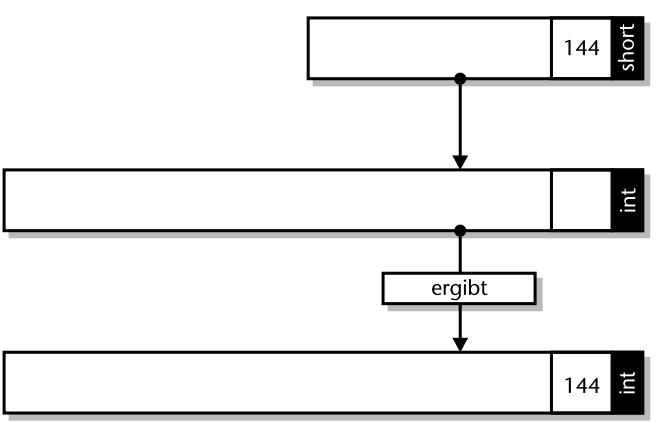


*Abbildung 4.1: Implizite Konvertierung von short nach Int32*

Wie aus der Abbildung zu erkennen, kann bei dieser impliziten Konvertierung kein Fehler auftreten. Der Zieldatentyp ist größer, kann also den Wert problemlos aufnehmen.

### 4.2.2 Explizite Konvertierung (Casting)

Ganz anders sieht es aus, wenn wir einen Wert vom Typ **int** in einen Wert vom Typ **short** konvertieren wollen. In diesem Fall ist es nicht ganz so einfach, denn der Compiler merkt natürlich, dass **int** einen größeren Wertebereich besitzt als **short**, es also zu einem Überlauf bzw. zu verfälschten Ergebnissen kommen könnte. Aus diesem Grund ist die folgende Zuweisung nicht möglich, auch wenn es von der Größe des Wertes her durchaus in Ordnung ist:

```
int i = 100;
short s;

s = i;
```

Der Compiler müsste in diesem Fall versuchen, einen großen Wertebereich in einem Datentyp mit einem kleineren Wertebereich unterzubringen. Als Vergleich: Er versucht, eine Literflasche Wasser in einem Schnapsglas unterzubringen.

***Casting***

Wir können nun aber dem Compiler sagen, dass der zu konvertierende Wert klein genug ist und dass er konvertieren soll. Eine solche Konvertierung wird als *explizite Konvertierung* oder auch als *Casting* bezeichnet. Der gewünschte Zieldatentyp wird in Klammern vor den zu konvertierenden Wert oder Ausdruck geschrieben:

```
int i = 100;
short s;

s = (short)i;
```

Jetzt funktioniert auch die Konvertierung. Aus Gründen der Übersichtlichkeit wird oftmals auch der zu konvertierende Wert in Klammern gesetzt, also

```
s = (short)(i);
```

Allgemein ausgedrückt: Verwenden Sie immer die *implizite Konvertierung*, wenn der Quelldatentyp einen kleineren Wertebereich besitzt als der Zieldatentyp, und die *explizite Konvertierung*, wenn der Zieldatentyp den kleineren Wertebereich besitzt. Achten Sie aber darauf, dass Sie die eigentlichen Werte nicht zu groß werden lassen.

Eine Umwandlung ist immer dann *implizit*, wenn kein Fehler auftreten kann, da der Wert des Quelldatentyps immer in den Wertebereich des Zieldatentyps passt. Eine Umwandlung wird als *explizit* bezeichnet, wenn beim Umwandlungsvorgang ein Fehler auftreten kann, weil der Zielbereich kleiner ist als der Wertebereich des Quelldatentyps.

### 4.2.3 Fehler beim Casting

Sie müssen natürlich auch beim Casting darauf achten, dass der eigentliche Wert in den Wertebereich des Zieldatentyps passt. Das ist Grundvoraussetzung, denn ansonsten hilft Ihnen auch ein Casting nicht weiter. Was passieren würde, wenn der Wert zu groß wäre, sehen Sie in Abbildung 4.2.

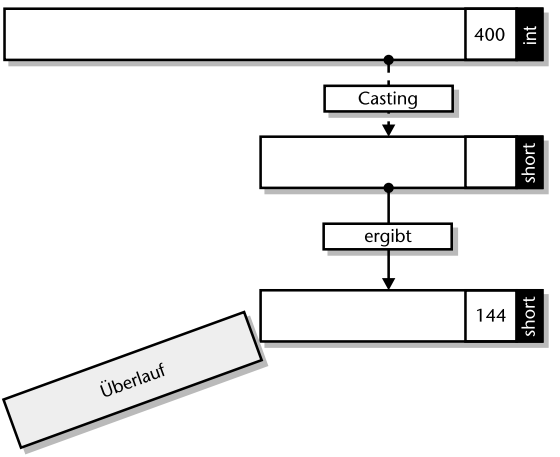


*Abbildung 4.2: Fehler beim Casting mit zu großem Wert*

C# würde allerdings keinen Fehler melden, lediglich der Wert wäre verfälscht. C# würde ebenso viele Bits in dem Zieldatentyp unterbringen, wie dort Platz haben, und die restlichen verwerfen. Wenn wir also den Wert 512 in einer Variablen vom Datentyp `sbyte` unterbringen wollten, der lediglich 8 Bit hat, ergäbe das den Wert 0.

***Konvertierungsfehler***

Um das genau zu verstehen, müssen Sie daran denken, dass der Computer lediglich mit Bits arbeitet, also mit 0 oder 1. Die unteren Bits des Werts werden problemlos in dem kleineren Datentyp untergebracht, während die oberen verloren gehen. Abbildung 4.3 veranschaulicht dies nochmals.

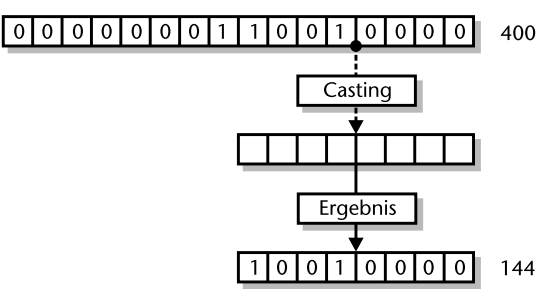


*Abbildung 4.3: Fehler beim Casting mit zu großem Wert (bitweise)*

### 4.2.4 Konvertierungsfehler erkennen

Fehler werden in C# durch Exceptions behandelt, die wir in Kapitel 11 behandeln werden. Hier geht es nicht darum, wie man eine solche Exception abfängt, sondern wie man C# dazu bringt, den Fehler beim Casting zu erkennen. Wie wir gesehen haben, funktioniert das nicht automatisch, wir müssen also ein wenig nachhelfen.

*checked*

Um bei expliziten Konvertierungen Fehler zu entdecken (und dann auch eine Exception auszulösen), verwendet man einen speziellen Anweisungsblock, den **checked**-Block. Nehmen wir ein Beispiel, bei dem der Anwender eine Integer-Zahl eingeben kann, die dann in einen Wert vom Typ **byte** umgewandelt wird. Der Zieldatentyp hat lediglich 8 Bit zur Verfügung, der Quelldatentyp liefert 32 Bit – Damit darf die Zahl nicht größer sein als 255, sonst schlägt die Konvertierung fehl. Für diesen Fall wollen wir vorsorgen und betten die Konvertierung daher in einen **checked**-Block ein:

```
/* Beispiel Typumwandlung (checked) 1 */
/* Autor:   Frank Eller               */
/* Sprache: C#                        */

using System;

public class Beispiel
{
  public static void Main()
  {
    int source = Console.ReadLine().ToInt32();
    sbyte target;
    checked
    {
      target = (byte)(source);
      Console.WriteLine("Wert: {0}",target);
    }
  }
}
```

Das Programm finden Sie auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_4\TYPUMWANDLUNG1.

Die Konvertierung wird nun innerhalb des **checked**-Blocks überwacht. Schlägt sie fehl, wird eine Exception ausgelöst (in diesem Fall `System.OverflowException`), die Sie wiederum abfangen können. Exceptions sind Ausnahmefehler, die ein Programm normalerweise beenden und die Sie selbst abfangen und auf die Sie reagieren können. Mehr über Exceptions erfahren Sie in Kapitel 10. Abbildung 4.4 ver-

deutlicht nochmals das Verhalten zur Laufzeit bei Verwendung eines **checked**-Blocks.

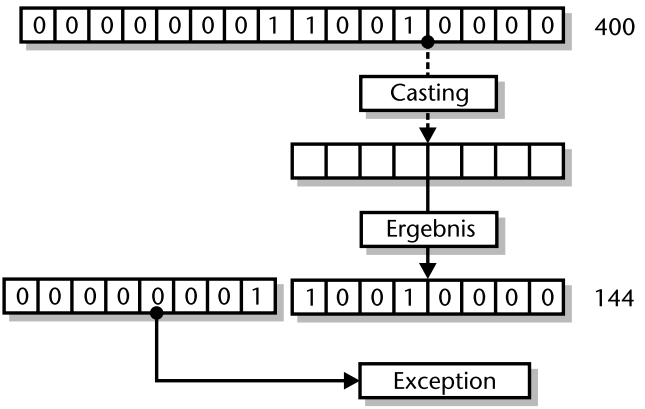


*Abbildung 4.4: Exception mittels checked-Block auslösen*

Die Überwachung wirkt sich aber nicht auf Methoden aus, die aus dem **checked**-Block heraus aufgerufen werden. Wenn Sie also eine Methode aufrufen, in der ebenfalls ein Casting durchgeführt wird, wird keine Exception ausgelöst. Stattdessen verhält sich das Programm wie oben beschrieben, der Wert wird verfälscht, wenn er größer ist als der maximale Bereich des Zieldatentyps. Im nächsten Beispiel wird dieses Verhalten verdeutlicht:

```
/* Beispiel Typumwandlung (checked) 2 */
/* Autor:   Frank Eller               */
/* Sprache: C#                        */

class TestClass
{
  public byte DoCast(int theValue)
  {
    //Casting von int nach Byte
    //falscher Wert, wenn theValue>255

    return (byte)(theValue);
  }

  public void Test(int a, int b)
  {
    byte v1;
    byte v2;

    checked
```

```
    {
      v1 = (byte)(a);
      v2 = DoCast(b);
    }
    Console.WriteLine("Wert 1: {0}\nWert 2: {1}",v1,v2);
  }
}

class Beispiel
{
  public static void Main()
  {
    int a,b;
    TestClass tst = new TestClass();

    Console.Write("Wert 1 eingeben: ");
    a = Console.ReadLine().ToInt32();
    Console.Write("Wert 2 eingeben: ");
    b = Console.ReadLine().ToInt32();

    tst.Test(a,b);
  }
}
```

Das Programm finden Sie auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_4\TYPUMWANDLUNG2.

Im Beispiel wird zweimal ein Casting durchgeführt, einmal direkt innerhalb des `checked`-Blocks und einmal in der Methode `Test()`, die aus dem `checked`-Block heraus aufgerufen wird. Wenn der erste Wert größer ist als 255, wird wie erwartet eine Exception ausgelöst. Nicht aber, wenn der zweite Wert größer ist. In diesem Fall wird die Umwandlung in der Methode `Test()` durchgeführt, eine Exception tritt nicht auf.

### 4.2.5 Umwandlungsmethoden

#### Methoden des Quelldatentyps

Wir haben nun gesehen, dass es problemlos möglich ist, Zahlenwerte in die verschiedenen numerischen Datentypen zu konvertieren. Aber was ist eigentlich, wenn wir beispielsweise eine Zahl in eine Zeichenkette konvertieren müssen, z.B. für eine Ausgabe oder weil die Methode, die wir benutzen wollen, eine Zeichenkette erwartet? Und wie sieht es umgekehrt aus, ist es auch möglich, eine Zeichenkette in eine Zahl umzuwandeln?

Ja, ist es. Aber das wissen Sie ja bereits, denn wir hatten es schon angesprochen.

In C# ist alles eine Klasse, auch die verschiedenen Datentypen sind nichts anderes als Klassen. Und als solche stellen sie natürlich Methoden zur Verfügung, die die Funktionalität beinhalten, mitunter auch Methoden für die Typumwandlung. Diese Methoden beginnen immer mit einem `To`, dann folgt der entsprechende Wert. Wenn Sie beispielsweise einen **string**-Wert in einen 32-Bit **int**-Wert konvertieren möchten (vorausgesetzt, die verwendete Zeichenkette entspricht einer ganzen Zahl), verwenden Sie die Methode `ToInt32()`:

```
string myString = "125";
int    myInt;

myInt = myString.ToInt32();
```

Umgekehrt funktioniert es natürlich auch, in diesem Fall verwenden Sie die Methode `ToString()`:

```
string myString;
int    myInt = 125;

myString = myInt.ToString();
```

Alle Umwandlungsmethoden finden Sie in der Tabelle 4.3. Diese Umwandlungsmethoden werden vom Datentyp **object** bereitgestellt, Sie finden sie daher in jedem Datentyp.

| Umwandlungsmethoden | | | |
|---|---|---|---|
| ToBoolean() | ToDate() | ToInt32() | ToString() |
| ToByte() | ToDecimal() | ToInt64() | ToUInt16() |
| ToChar() | ToDouble() | ToSByte() | ToUInt32() |
| ToDateTime() | ToInt16() | ToSingle() | ToUInt64() |
| ToBoolean() | | | |

*Tabelle 4.3: Die Umwandlungsmethoden der Datentypen*

Bei allen Umwandlungen, ob es nun durch die entsprechenden Methoden, durch implizite Umwandlung oder durch Casting geschieht, ist immer der verwendete Datentyp zu beachten. So ist es durchaus möglich, einen Gleitkommawert in eine ganze Zahl zu konvertieren, man muss sich allerdings darüber im Klaren sein, dass dadurch die Genauigkeit verloren geht. Ebenso sieht es aus, wenn man z.B. einen Wert vom Typ **decimal** in den Datentyp **double** umwandelt, der weniger genau ist. Auch hier ist die Umwandlung zwar möglich, die Genauigkeit geht allerdings verloren.

### Methoden des Zieldatentyps

*Parse()*

Die Umwandlung eines String in einen anderen Zieldatentyp, z.B. **int** oder **double**, funktioniert auch auf einem anderen Weg. Die numerischen Datentypen bieten hierfür die Methode `Parse()` an, die in mehreren überladenen Versionen existiert und grundsätzlich die Umwandlung eines String in den entsprechenden Datentyp veranlassen. Der Vorteil der Methode `Parse()` ist, dass zusätzlich noch angegeben werden kann, wie die Zahlen formatiert sind bzw. in welchem Format sie vorliegen. Ein einfaches Beispiel für `Parse()` liefert uns die Methode `Main()`, die es uns ermöglicht, auch Parameter zu übergeben:

```
/* Beispiel Wertumwandlung 1 */
/* Autor:   Frank Eller      */
/* Sprache: C#               */

using System;

public class Beispiel
{
  public static int Main(string[] args)
  {
    // Ermitteln des ersten Zahlenwerts
    int FirstValue = 0;
    FirstValue = Int32.Parse(args[0]);

    // ... weitere Anweisungen ...
  }
```

Das Array `args[]` enthält die an das Programm in der Kommandozeile übergebenen Parameter. Was es mit Arrays auf sich hat, werden wir in Kapitel 7 noch ein wenig näher betrachten, für den Moment soll genügen, dass es sich dabei um ein Feld mit Daten handelt, die alle den gleichen Typ haben und über einen Index angesprochen werden können.

Andere Programmiersprachen besitzen ebenfalls die Möglichkeit, Parameter aus der Kommandozeile an das Programm zu übergeben. Es gibt jedoch einen Unterschied: In C# wird der Name des Programms nicht mit übergeben, d.h. das erste Argument, das Sie in `Main` auswerten können, ist auch wirklich der erste übergebene Kommandozeilenparameter.

Im obigen Fall erwartet das Programm eine ganze Zahl vom Typ **int**. Die Methode `Parse()` wird benutzt, um den String in einen Integer umzuwandeln. Dabei hat diese Methode wie ebenfalls schon angesprochen den Vorteil, nicht nur den Zahlenwert einfach so zu kon-

vertieren, sondern auch landesspezifische Einstellungen zu berücksichtigen. Für die herkömmliche Konvertierung ist die Methode `ToInt32()` des Datentyps **string** absolut ausreichend:

```
/* Beispiel Wertumwandlung 2 */
/* Autor:   Frank Eller      */
/* Sprache: C#               */

using System;

public class Beispiel
{
  public static int Main(string[] args)
  {
    // Ermitteln des ersten Zahlenwerts
    int FirstValue = 0;
    FirstValue = args[0].ToInt32();

    // ... weitere Anweisungen ...
  }
```

Der Datentyp **string** ist außerdem ein recht universeller Datentyp, der sehr häufig in Programmen verwendet wird. Aus diesem Grund werden wir uns den Strings in einem gesonderten Abschnitt zuwenden.

## 4.3 Boxing und Unboxing

Wir haben bereits gelernt, dass es zwei Arten von Daten gibt, Referenztypen und Wertetypen. Möglicherweise haben Sie sich bereits gefragt, warum man mit Wertetypen ebenso umgehen kann wie mit Referenztypen, wo es sich doch um zwei unterschiedliche Arten des Zugriffs handelt bzw. die Daten auf unterschiedliche Art im Speicher des Computers abgelegt sind. Der Trick bzw. das Feature, das C# hier verwendet, heißt *Boxing*.

***Boxing***

Wenn ein Wertetyp als Referenztyp verwendet werden soll, werden die enthaltenen Daten sozusagen verpackt. C# benutzt dafür den Datentyp **object**, der bekanntlich die Basis aller Datentypen darstellt und somit auch jeden Datentyp aufnehmen kann. Im Unterschied zu anderen Sprachen merkt sich **object** aber, welcher Art von Daten in ihm gespeichert sind, um eine Konvertierung in die andere Richtung ebenfalls zu ermöglichen.

Mit diesem Objekt, bei dem es sich nun um einen Referenztyp handelt, ist das Weiterarbeiten problemlos möglich. Umgekehrt können Sie einen auf diese Art und Weise umgewandelten Wert auch wieder in einen Wertetyp zurückkonvertieren.

*Automatisches Boxing*

Das einfachste Beispiel haben wir bereits kennen gelernt, nämlich die Methode `WriteLine()` der Klasse `Console`. Dieser Methode können wir übergeben, was wir wollen, sei es ein Referenztyp, ein Wertetyp oder einfach nur den Wert selbst – `WriteLine()` nimmt das Angebot anstandslos an und gibt die Daten auf dem Bildschirm aus.

Der Datentyp der Parameter von `WriteLine()` ist **`object`**. Intern verwendet C# nun das Boxing, um eine Hülle um den übergebenen Wert zu legen und ihn auf diese Art und Weise als Referenztyp behandeln zu können. Dieses automatische Boxing tritt überall auf, wo ein Wertetyp übergeben, aber ein Referenztyp benötigt wird.

### 4.3.1 Boxing

Sie können Boxing und das Gegenstück Unboxing auch selbst in Ihren Applikationen anwenden. Der folgende Code speichert den Wert einer **`int`**-Variable in einer Variable vom Typ **`object`**, also einem Referenztyp.

```
/* Beispiel Boxing 1    */
/* Autor:   Frank Eller */
/* Sprache: C#          */

using System;

public class TestClass
{
  public static void Main()
  {
    int i = 100;
    object o;
    o = i;  //Boxing !!
    Console.WriteLine("Wert ist {0}.",o);
  }
}
```

Der Wert von `i` ist nun in einem Objekt `o` gespeichert. Grafisch dargestellt sieht das dann so aus, wie in Abbildung 4.5. Die Ausgabe des Programms entspricht der Ausgabe des Wertes von `i`:

**Wert ist 100.**

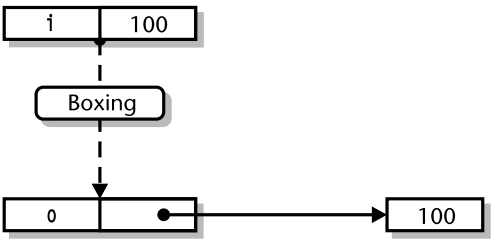


*Abbildung 4.5: Boxing eines Integer-Werts*

Das Beispielprogramm finden Sie auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_4\BOXING1.

### 4.3.2 Unboxing

Der umgekehrte Weg ist zwar vom Prinzip her ebenso einfach, allerdings muss der Datentyp, in den das Objekt zurückkonvertiert werden soll, bekannt sein. Es ist nicht möglich, ein Objekt, das einen **int**-Wert enthält, in einen **byte**-Wert umzuwandeln. Hier zeigt sich wieder die Typsicherheit von C#.

```
/* Beispiel Boxing 2    */
/* Autor:   Frank Eller */
/* Sprache: C#          */

using System;

public class TestClass
{
  public static void Main()
  {
    int i = 100;
    object o;
    o = i;  //Boxing !!
    Console.WriteLine("Wert ist {0}.",o);

    //Rückkonvertierung
    byte b = (byte)(o); //funktioniert nicht!!
    Console.WriteLine("Byte-Wert: {0}",b);
  }
}
```

Den Quellcode finden Sie auf der beiligenden CD im Verzeichnis BEISPIELE\KAPITEL_4\BOXING2.

Obwohl die Größe des in `o` enthaltenen Werts durchaus in eine Variable vom Typ **byte** passen würde, ist dieses Unboxing nicht möglich.

Im Objekt o ist der enthaltene Datentyp mit gespeichert, damit verlangt C# beim Unboxing, dass auch hier ein int-Wert für die Rückkonvertierung verwendet wird.

Wir haben jedoch bereits die andere Möglichkeit der Typumwandlung, die explizite Umwandlung oder das Casting, kennen gelernt. Wenn eine implizite Konvertierung nicht funktioniert, sollte es doch eigentlich mit einer expliziten Konvertierung funktionieren. Das folgende Beispiel beweist dieses.

```
/* Beispiel Boxing 3    */
/* Autor:   Frank Eller */
/* Sprache: C#          */

using System;

public class TestClass
{
  public static void Main()
  {
    int i = 100;
    object o;
    o = i;  //Boxing !!
    Console.WriteLine("Wert ist {0}.",o);

    //Rückkonvertierung
    byte b = (byte)((int)(o)); //funktioniert!!
    Console.WriteLine("Byte-Wert: {0}.",b);
  }
}
```

Das Beispielprogramm finden Sie auf der beiligenden CD im Verzeichnis BEISPIELE\KAPITEL_4\BOXING3.

In diesem Beispiel wird der in o enthaltene Wert zunächst in einen int-Wert zurückkonvertiert, wonach aber unmittelbar das Casting zu einem byte-Wert folgt. Und da der enthaltene Wert nicht zu groß für den Datentyp byte ist, ergibt sich als Ausgabe:

**Wert ist 100.**

**Byte-Wert: 100.**

Beim Boxing wird ein Wertetyp in einen Referenztyp „verpackt“. Anders als in diversen anderen Programmiersprachen merkt sich das Objekt in C# aber, welcher Datentyp darin verpackt wurde. Damit ist ein Unboxing nur in den gleichen Datentyp möglich.

### 4.3.3 Den Datentyp ermitteln

Der im Objekt `o` enthaltene Datentyp kann auch ermittelt werden. `o` stellt dafür die Methode `GetType()` zur Verfügung, die den Typ der enthaltenen Daten zurückliefert. Der Ergebnistyp von `GetType()` ist `Type`. Und da `Type` auch eine Methode `ToString()` enthält, ist es mit folgender Konstruktion möglich, den Datentyp als **string** auszugeben:

```
/* Beispiel Boxing 4    */
/* Autor:   Frank Eller */
/* Sprache: C#          */

using System;

public class TestClass
{
  public static void Main()
  {
    int i = 100;
    object o;
    o = i;  //Boxing !!
    Console.WriteLine("Wert ist {0}.",o);
    Console.WriteLine(o.GetType().ToString());
  }
}
```

Das Beispielprogramm finden Sie auf der beiligenden CD im Verzeichnis BEISPIELE\KAPITEL_4\BOXING2.

Damit wäre das Boxing soweit abgehandelt. Normalerweise werden Sie es in Ihren Applikationen dem .net-Framework überlassen, das Boxing durchzuführen. Es funktioniert ja auch automatisch und problemlos. Manuelles Boxing oder Unboxing ist in den seltensten Fällen nötig, aber wie Sie sehen auch nicht besonders schwierig.

## 4.4 Strings

Der Datentyp **string** ist ein recht universell einsetzbarer Datentyp, den wir auch schon in einem Beispiel benutzt haben. Strings sind Zeichenketten, d.h. eine Variable von Typ **string** kann jedes beliebige Zeichen aufnehmen. Weiterhin bietet auch dieser Datentyp mehrere Funktionen zum Arbeiten mit Zeichenketten.

**string** weist auch noch eine andere Besonderheit auf. Obwohl die Deklaration wie bei einem Wertetyp funktioniert, handelt es sich doch um einen Referenztyp, denn eine Variable vom Typ **string**

kann so viele Zeichen aufnehmen, wie Platz im Speicher ist. Damit ist die Größe einer `string`-Variablen nicht festgelegt, der verwendete Speicher muss dynamisch (auf dem Heap) reserviert werden. Der Datentyp `string` ist (zusammen mit `object`) der einzige Basisdatentyp, der ein Referenztyp ist. Alle anderen Basistypen sind Wertetypen.

### 4.4.1 Unicode und ASCII

*ASCII*

Der ASCII-Zeichensatz (American Standard Code for Information Interchange) war der erste Zeichensatz auf einem Computer. Anfangs arbeitete man noch mit einem 7-Bit-ASCII-Zeichensatz, wodurch 127 Zeichen darstellbar waren. Das genügte für alle Zeichen des amerikanischen Alphabets. Später jedoch wurde der Zeichensatz auf 8 Bit Breite ausgebaut, um die Sonderzeichen der meisten europäischen Sprachen ebenfalls aufnehmen zu können, und für die meisten Anwendungen genügte dies auch. Unter Windows konnte man sich den Zeichensatz aussuchen, der für das entsprechende Land passend war, und ihn benutzen.

In Zeiten, da das Internet eine immer größere Rolle spielt, und zwar sowohl bei der Informationsbeschaffung als auch bei der Programmierung, genügt ein Byte nicht mehr, um alle Zeichen darzustellen. Genauer gesagt: Wenn jemand auf eine Internet-Seite zugreifen will, muss dafür auch der Zeichensatz installiert sein, mit dem diese Seite arbeitet. Uns als Europäern fällt das nicht besonders auf, meist bewegen wir uns auf deutschen oder englischen Seiten, bei denen der Zeichensatz ohnehin zum größten Teil übereinstimmt. Was aber, wenn wir auf eine japanische oder chinesische Seite zugreifen wollen? In diesem Fall sehen wir auf dem Bildschirm nicht die entsprechenden Schriftzeichen, sondern in den meisten Fällen einen Mischmasch aus Sonderzeichen ohne irgendetwas lesen zu können. Um es noch genauer zu sagen: Auch ein Chinese hätte durchaus Probleme, seine Sprache wiederzuerkennnen.

*Unicode*

C# wurde von Microsoft als eine Sprache angekündigt, die die Anwendungsentwicklung sowohl für das Web als auch für lokale Computer vereinfachen soll. Gerade bei der Entwicklung von Internetapplikationen ist es aber sehr wichtig, dass es keine Konflikte mit dem Zeichensatz gibt. Deshalb arbeitet C# komplett mit dem *Unicode*-Zeichensatz, bei dem ein Zeichen nicht durch ein Byte, sondern durch zwei Byte repräsentiert wird.

Der Unterschied ist größer, als man denkt. Waren mit 8 Bit noch $2^7$ Zeichen (= 255 Zeichen) darstellbar, sind es jetzt $2^{15}$ Zeichen (= 65.535 Zeichen). Diese Anzahl genügt, um alle Zeichen aller Sprachen dieser

Welt und noch einige Sonderzeichen unterzubringen. Um die Größenordnung noch deutlicher darzustellen: Etwa ein Drittel des Unicode-Zeichensatzes sind noch unbelegt.

C# arbeitet komplett mit dem Unicode-Zeichensatz. Sowohl was die Strings innerhalb Ihres eigenen Programms angeht als auch was die Quelltexte betrifft, auch hier wird der Unicode-Zeichensatz verwendet, theoretisch ist also jedes Zeichen darstellbar. Allerdings gilt für die Programmierung nach wie vor nur der englische (bzw. amerikanische) Zeichensatz mit den bekannten Sonderzeichen. Eine Variable mit dem Bezeichner *Zähler* ist leider nicht möglich. Der Grund hierfür ist allerdings auch offensichtlich: Immerhin soll mit der Programmiersprache in jedem Land gearbeitet werden können, somit muss man einen kleinsten Nenner finden. Und bezüglich des Zeichensatzes ist das nun mal der amerikanische Zeichensatz.

### 4.4.2 Standard-Zuweisungen

Zeichenketten werden immer in doppelten Anführungszeichen angegeben. Die folgenden Zuweisung an eine Variable vom Datentyp **string** wäre also der Normalfall:

```
string myString = "Hallo Welt";
```

oder natürlich

```
string myString;
myString = "Hallo Welt";
```

Es gibt aber noch eine weitere Möglichkeit, einem String einen Wert zuzuweisen. Wenn Sie den Inhalt eines bereits existierenden String kopieren möchten, können Sie die statische Methode `Copy()` verwenden und den Inhalt eines bestehenden String an den neuen String zuweisen:

```
string myString = "Frank Eller";
string myStr2   = string.Copy(myString);
```

Ebenso ist es möglich, nur einen Teilstring zuzuweisen. Dazu wird eine Instanzmethode des erzeugten Stringobjekts verwendet:

```
string myString = "Frank Eller";
string myStr2   = myString.Substring(6);
```

Die Methode `Substring()` kopiert einen Teil des bereits bestehenden String `myString` in den neu erstellten `myStr2`. `Substring()` ist eine überladene Methode, Sie können entweder den Anfangs- und Endpunkt der Kopieraktion angeben oder nur den Anfangspunkt, also den Index

des Zeichens, bei dem die Kopieraktion begonnen werden soll. Denken Sie daran, dass immer bei 0 mit der Zählung begonnen wird, d.h. das siebte Zeichen hat den Index 6. Wenn Sie die zweite Variante benutzen, wird der gesamte String bis zum Ende kopiert.

Zusätzlich zu diesen Möglichkeiten gibt es noch erweiterte Zuweisungsmöglichkeiten. Ebenso wie bei der Ausgabe durch `WriteLine()` gelten z.B. auch bei Strings die Escape-Sequenzen, denn `WriteLine()` tut ja nichts anderes, als den String, den Sie angeben, zu interpretieren und auszugeben.

### 4.4.3 Erweiterte Zuweisungsmöglichkeiten

*Escape-Sequenzen*

Kommen wir hier zunächst zu den bereits angesprochenen Escape-Sequenzen. Diese können natürlich auch hier vollständig benutzt werden. So können Sie z.B. auf folgende Art einen String dazu bringen, doppelte Anführungszeichen auszugeben:

```
string myString = "Dieser Text hat \"Anführungszeichen\".";
Console.WriteLine(myString);
```

Die Ausgabe wäre dann entsprechend:

**Dieser Text hat "Anführungszeichen".**

*Literalzeichen*

Alle anderen Escape-Sequenzen, die Sie bereits kennen gelernt haben, sind ebenfalls möglich. Allerdings benötigen Sie diese nicht, um Sonderzeichen darstellen zu können. In C# haben Sie Strings betreffend noch eine weitere Möglichkeit, nämlich die, die Escape-Sequenzen nicht zu bearbeiten. Ein Beispiel soll deutlich machen, wozu dies gut sein kann.

Nehmen wir an, Sie wollten einen Pfad zu einer bestimmten Datei in einem String speichern. Das kommt durchaus öfter vor, z.B. wenn Sie in Ihrem Programm die letzte verwendete Datei speichern wollen. Sobald Sie jedoch den Backslash als Zeichen benutzen, wird das von C# als Escape-Sequenz betrachtet, woraus folgt, dass Sie für jeden Backslash im Pfad eben zwei Backslashes hintereinander schreiben müssen:

```
string myString = "d:\\aw\\csharp\\Kapitel5\\Kap05.doc";
```

*Das @-Zeichen*

Einfacher wäre es, wenn in diesem Fall die Escape-Sequenzen nicht bearbeitet würden, wir also den Backslash nur einmal schreiben müssten. Das würde im Übrigen auch der normalen Schreibweise entsprechen. Immerhin können wir nicht verlangen, wenn ein Anwender einen Dateinamen eingibt, dass dieser jeden Backslash doppelt schreibt. Um die Bearbeitung der Escape-Sequenzen zu verhin-

dern schreiben wir vor den eigentlichen String einfach ein @-Zeichen:

```
string myString = @"d:\aw\csharp\Kapitel5\Kap05.doc";
```

Fortan werden die Escape-Sequenzen nicht mehr bearbeitet, es genügt jetzt, einen Backslash zu schreiben.

***Sonderzeichen***

Sie werden sich möglicherweise fragen, wie Sie in einem solchen String ohne Escape-Sequenz z.B. ein doppeltes Anführungszeichen schreiben. Denn die oben angesprochene Möglichkeit existiert ja nicht mehr, der Backslash würde als solcher angesehen und das darauf folgende doppelte Anführungszeichen würde das Ende des String bedeuten. Die Lösung ist ganz einfach: Schreiben Sie solche Sonderzeichen einfach doppelt:

```
string myString = "Das sind ""Anführungszeichen"".";
```

### 4.4.4 Zugriff auf Strings

Es gibt mehrere Möglichkeiten, auf einen String zuzugreifen. Die eine Möglichkeit besteht darin, den gesamten String zu benutzen, wie wir es oftmals tun. Eine weitere Möglichkeit, die wir auch schon kennen gelernt haben, ist die, auf einen Teilstring zuzugreifen (mittels der Methode `Substring()`). Es existiert aber noch eine Möglichkeit.

Strings sind Zeichenketten. Wenn man diesen Begriff wörtlich nimmt, sind Strings tatsächlich aneinander gereihte Zeichen. Der Datentyp für ein Zeichen ist `char`. Damit kann auf einen String auch zeichenweise zugegriffen werden.

Die Eigenschaft `Length` eines String liefert dessen Länge zurück. Wir könnten also eine `for`-Schleife benutzen, alle Zeichen eines String zu kontrollieren. Die `for`-Schleife haben wir zwar noch nicht behandelt, in diesem Fall werde ich aber dem entsprechenden Kapitel ein wenig vorgreifen und die `for`-Schleife hier schon benutzen. Auf die genaue Funktionsweise werden wir in Kapitel 5 noch eingehen.

Mit Hilfe der `for`-Schleife können wir einen Programmblock mehrfach durchlaufen. Zum Zählen wird eine Variable benutzt, die wir dazu verwenden können, jedes Zeichen des String einzeln auszuwerten.

```
/* Beispiel Stringzugriff 1 */
/* Autor:   Frank Eller     */
/* Sprache: C#              */

using System;

class TestClass
{
  public static void Main()
  {
    string myStr = "Hallo Welt.";
    string xStr  = "";

    for (int i=0;i<myStr.Length;i++)
    {
      string x = myStr[i].ToString();
      if (!(x=="e"))
        xStr += x;
    }

    Console.WriteLine(xStr);
  }
}
```

Wenn Sie dieses Programm ausführen, ergibt sich als Ausgabe

**Hallo Welt.**

Sie finden auch dieses Programm auf der beiliegenden CD, im Verzeichnis BEISPIELE\KAPITEL_4\STRINGS1.

Wir kontrollieren jedes Zeichen des String darauf, ob es sich um ein „e" handelt. Ist dies der Fall, tun wir nichts, ansonsten fügen wir den Buchstaben unserem zweiten String hinzu. Da die Abfrage `myStr[i]` den Datentyp **char** zurückliefert, müssen wir diesen vor der Kontrolle in den Datentyp **string** umwandeln. Da dies durch Casting nicht möglich ist, verwenden wir die Methode `ToString()` des Datentyps **char**.

Auch hier bemerken Sie wieder, dass C# eine typsichere Sprache ist. In anderen Sprachen ist es durchaus möglich, die Datentypen **char** und **string** zu mischen, da ein String meist nur eine Kette aus Chars darstellt. In C# werden diese beiden Datentypen allerdings vollkommen unterschiedlich behandelt.

***Rechenoperatoren***

Sicherlich haben Sie außerdem im obigen Beispiel bemerkt, dass hier ein Rechenoperator auf einen String angewendet wurde, nämlich der Operator +=. Tatsächlich ist es so, dass der +-Operator sowie der +=-Operator tatsächlich auf Strings angewendet werden können und zwei Strings zu einem zusammenfügen:

```
string myString1 = "Frank";
string myString2 = " Eller";
string myString  = myString1+myString2;
```

In diesem Fall würde `myString` den Wert „Frank Eller“ enthalten.

### 4.4.5 Methoden von string

Der Datentyp **string** ist wie jeder andere Datentyp in C# auch eine Klasse und stellt somit einige Methoden zur Verfügung, die Sie im Zusammenhang mit Zeichenketten verwenden können. In den beiden folgenden Listen werden einige häufig verwendete Methoden des Datentyps **string** vorgestellt.

#### Klassenmethoden von string

Die Klassenmethoden von **string** sind allesamt überladene Methoden mit relativ vielen Aufrufmöglichkeiten. An dieser Stelle werde ich mich daher auf die wichtigsten bzw. am häufigsten verwendeten Methoden beschränken.

```
public static bool Compare(
    string strA
    string strB
);
public static bool Compare(
    string strA
    string strB
    bool ignoreCase
);
```

Mit diesen Versionen der Methode `Compare()` können zwei komplette Zeichenketten miteinander verglichen werden. Mit dem Parameter `ignoreCase` können Sie angeben, ob die Groß-/Kleinschreibung ignoriert werden soll oder nicht.

```
public static bool Compare(
    string strA;
    int indexA;
    string strB;
    int indexB;
    int length
);
public static bool Compare(
    string strA;
    int indexA;
    string strB;
```

```
    int indexB;
    int length;
    bool ignoreCase
);
```

Mit diesen Versionen der Methode `Compare()` können Sie vergleichen, ob bestimmte Teile zweier Zeichenketten übereinstimmen. Auch hier ist es wieder möglich, die Groß-/Kleinschreibung zu ignorieren.

```
public static string Concat(object);
public static string Concat(string[] values);
public static string Concat(object[] args);
```

Die Methode `Concat()` dient dem Zusammenfügen mehrerer Zeichenketten bzw. Objekte, die Zeichenketten repräsentieren.

```
public static string Copy(string str0);
```

Die Methode `Copy()` liefert eine Kopie des übergebenen String zurück.

```
public static bool Equals(
    string a,
    string b
);
```

Die Methode `Equals()` kontrolliert, ob die beiden übergebenen Strings gleich sind. Auf Groß-/Kleinschreibung wird bei diesem Vergleich geachtet. Sind beide Strings gleich, wird **true** zurückgeliefert, sind sie es nicht oder ist einer der übergebenen Strings ohne Zuweisung (**null**), wird **false** zurückgeliefert.

```
public static string Format(
    string format,
    object arg0
);
public static string Format(
    string format,
    object[] args
);
```

Die Methode `Format()` ermöglicht die Formatierung von Werten. Dabei enthält der übergebene Parameter `format` den zu formatierenden String mit Platzhaltern und Formatierungszeichen, der Parameter `arg0` enthält das Objekt, das an der Stelle des Platzhalters eingefügt und entsprechend der Angaben formatiert wird. Mehr über die `Format()`-Funktion erfahren Sie in Kapitel 4.5.

### Instanzmethoden von string

```
public object Clone();
```

Die Methode `Clone()` liefert die aktuelle String-Instanz als Objekt zurück.

```
public int CompareTo(object o);
public int CompareTo(string s);
```

Die Methode `CompareTo()` vergleicht die aktuelle String-Instanz mit dem als Parameter übergebenen Objekt bzw. String. Zurückgeliefert wird ein Integer-Wert, der angibt, wie die beiden Strings sich zueinander verhalten. Grundlage für den Vergleich ist das Alphabet, wobei ein String als umso kleiner angesehen wird, je weiter er im Vergleich Richtung Anfang angeordnet würde. Der Rückgabewert ist kleiner 0, wenn die aktuelle String-Instanz kleiner als der Parameter ist, gleich 0 wenn sie gleich dem Parameter ist, und größer 0 oder 1, wenn sie größer als der Parameter ist.

```
public bool EndsWith(string value);
```

Die Methode `EndsWith()` kontrolliert, ob der übergebene Parameter dem Ende der aktuellen String-Instanz entspricht. Wenn `value` länger ist als die aktuelle Instanz oder nicht dem letzten Teil entspricht, wird `false` zurückgeliefert.

```
public new bool Equals(string value);
public override bool Equals(object obj);
```

Die Methode `Equals()` existiert auch als Instanzmethode, funktioniert aber genauso wie die entsprechende statische Methode. Allerdings ist in diesem Fall der erste String, mit dem verglichen werden soll, bereits durch die aktuelle Instanz vorgegeben.

```
public Type GetType();
```

Die Methode `GetType()` ist in allen Klassen enthalten. Sie liefert den Datentyp zurück.

```
public int IndexOf(char[] value);
public int IndexOf(string value);
public int IndexOf(char value);
public int IndexOf(
    string value,
    int startIndex
);
public int IndexOf(
    char[] value,
    int startIndex
);
```

```
public int IndexOf(
    char value,
    int startIndex
);
public int IndexOf(
    string value,
    int startIndex,
    int endIndex
);
public int IndexOf(
    char[] value,
    int startIndex,
    int endIndex
);
public int IndexOf(
    char value,
    int startIndex,
    int endIndex
);
```

Die Methode `IndexOf()` ermittelt den Offset, an dem der angegebene Teilstring in der aktuellen String-Instanz auftritt. Wenn der Teilstring überhaupt nicht enthalten ist, wird der Wert –1 zurückgeliefert. Mit den Parametern `startIndex` bzw. `endIndex` kann auch noch ein Bereich festgelegt werden, in dem die Suche stattfinden soll.

Analog zur Methode `IndexOf()` existiert auch noch eine Methode `LastIndexOf()`, die das letzte Auftreten des angegebenen Teilstring zurückliefert. Sie existiert in den gleichen überladenen Versionen wie die Methode `IndexOf()`.

```
public string Insert(
    int startIndex,
    string value
);
```

Die Methode `Insert()` fügt einen Teilstring an der angegebenen Stelle in den aktuellen String ein und liefert das Ergebnis als **`string`** zurück.

```
public string PadLeft(int totalWidth);
public string PadLeft(
    int totalWidth,
    char paddingChar
);
```

Die Methode PadLeft() richtet einen String rechtsbündig aus und füllt ihn von vorne mit Leerstellen, bis die durch den Parameter totalLength angegebene Gesamtlänge erreicht ist. Falls gewünscht, kann mit dem Parameter paddingChar auch ein Zeichen angegeben werden, mit dem aufgefüllt wird.

```
public string PadRight(int totalWidth);
public string PadRight(
    int totalWidth,
    char paddingChar
);
```

Die Methode PadRight() verhält sich wie die Methode PadLeft(), nur dass der String jetzt linksbündig ausgerichtet wird.

```
public string Remove(
    int startIndex,
    int Count
);
```

Die Methode Remove() entfernt einen Teil aus der aktuellen String-Instanz. Die Anfangsposition und die Anzahl der Zeichen, die entfernt werden sollen, können Sie mit den Parametern startIndex und count angeben.

```
public string Replace(
    char oldChar,
    char newChar
);
```

Die Methode Replace() ersetzt im gesamten aktuellen String ein Zeichen gegen ein anderes.

```
public string[] Split(char[] separator);
public string[] Split(
    char[] separator,
    int count
);
```

Die Methode Split() teilt einen String in diverse Teilstrings auf. Als Trennzeichen wird das im Parameter separator angegebene Array aus Zeichen benutzt. Mit dem Parameter count können Sie zusätzlich die maximale Zahl an Teilstrings, die zurückgeliefert werden sollen, angeben.

```
public bool StartsWith(string value);
```

Die Methode StartsWith() kontrolliert, ob die aktuelle String-Instanz mit dem im Parameter value angegebenen Teilstring beginnt.

```
public string SubString(int startIndex)
public string SubString(
    int startIndex
    int length
);
```

Die Methode `Substring()` liefert einen Teilstring der aktuellen String-Instanz zurück, wobei Sie die Anfangsposition und die Länge angeben können. Wird die Länge nicht angegeben, geht der Teilstring bis zum Ende.

```
public string Trim()
public string Trim(char[] trimChars)
```

Die Methode `Trim()` entfernt standardmäßig alle Leerzeichen am Anfang und am Ende der aktuellen String-Instanz und liefert den resultierenden Teilstring zurück. Wollen Sie statt der Leerzeichen andere Zeichen entfernen, können Sie diese im Parameter `trimChars` angeben.

```
public string TrimEnd(char[] trimChars);
```

Die Methode `TrimEnd()` entfernt alle angegebenen Zeichen am Ende des String. Wenn Sie für den Parameter `trimChars` den Wert **null** übergeben, werden alle Leerzeichen entfernt.

```
public string TrimStart(char[] trimChars)
```

Die Methode `TrimStart()` entfernt alle im Array `trimChars` angegebenen Zeichen am Anfang der aktuellen String-Instanz und liefert die resultierende Zeichenkette zurück.

Weiterhin kommen natürlich noch die Methoden zum Konvertieren hinzu, die jeder Datentyp anbietet, wie in diesem Fall z.B. `ToBoolean()`, `ToInt32()`, `ToDateTime()` usw. Diese Umwandlungsmethoden sind bekanntlich in jedem Datentyp enthalten, so auch im Datentyp **string**.

## 4.5 Formatierung von Daten

### 4.5.1 Standardformate

Sie haben bereits im *Hallo-Welt*-Programm mit den Platzhaltern gearbeitet. Was ausgegeben wurde, war nichts anderes als eine feste Zeichenkette, also im Prinzip auch nur ein String. Sie haben allerdings über diese Platzhalter auch die Möglichkeit, die Ausgabe numerischer Werte zu formatieren.

**Format**

Normalerweise geschieht das durch die Methode `Format()`, die jeder Datentyp zur Verfügung stellt. Da die Formatierung sich immer darauf bezieht, Daten in einem String auszugeben, ruft die Methode `Format()` des Datentyps **string** auch immer die entsprechende Methode des verwendeten numerischen Datentyps auf, der formatiert werden soll. Und nicht nur das, sogar die Methode `WriteLine()` ruft `Format()` auf, wenn es nötig ist.

**Formatzeichen**

Die Angabe, welche Art von Formatierung gewünscht ist, geschieht im Platzhalter durch die Angabe eines Formatzeichens und ggf. einer Präzisionsangabe für die Anzahl der Stellen, die ausgegeben werden sollen. Ein Beispiel soll verdeutlichen, wie das funktioniert. Wir wollen zwei eingegebene Integer-Werte so formatieren, dass sie korrekt untereinander stehen. Dazu geben wir im Platzhalter an, dass alle numerischen Werte mit fünf Stellen ausgegeben werden sollen. Nimmt eine Zahl die Stellen nicht komplett ein, wird mit Nullen aufgefüllt.

```
/* Beispiel Formatierung 1 */
/* Autor:   Frank Eller    */
/* Sprache: C#             */

using System;

public class Beispiel
{
  public static void Main()
  {
    int a,b;
    Console.Write("Geben Sie Zahl 1 ein: ");
    a = Console.ReadLine().ToInt32();
    Console.Write("Geben Sie Zahl 2 ein: ");
    b = Console.ReadLine().ToInt32();
    Console.WriteLine("Die Zahlen lauten:");
    Console.WriteLine("Zahl 1: {0:D5}",a);
    Console.WriteLine("Zahl 2: {0:D5}",b);
  }
}
```

Bei einer Angabe zweier Zahlen 75 und 1024 würde die Ausgabe folgendermaßen aussehen:

Die Zahlen lauten
Zahl 1: 00075
Zahl 2: 01024

Die Zahlen stehen damit exakt untereinander. Das Beispielprogramm finden Sie auf der CD im Verzeichnis
BEISPIELE\KAPITEL_4\FORMATIERUNG1.

Die Angabe der Präzision hat jedoch unterschiedliche Bedeutung. Im Falle einer Gleitkommazahl würde nämlich nicht die Anzahl der Gesamtstellen angegeben, sondern die Anzahl der Nachkommastellen. Falls notwendig, rundet C# hier auch automatisch auf oder ab, um die geforderte Genauigkeit zu erreichen. Im Falle einer Hexadezimalausgabe würde eine ganze Zahl auch automatisch in das Hexadezimal-Format umgewandelt. Das Formatierungszeichen für das Hexadezimalformat ist *X*:

```
/* Beispiel Formatierung 2 */
/* Autor:   Frank Eller    */
/* Sprache: C#             */

using System;

public class Beispiel
{
  public static void Main()
  {
    int a,b;
    Console.Write("Geben Sie Zahl 1 ein: ");
    a = Console.ReadLine().ToInt32();
    Console.Write("Geben Sie Zahl 2 ein: ");
    b = Console.ReadLine().ToInt32();
    Console.WriteLine("Die Zahlen lauten:");
    Console.WriteLine("Zahl 1: {0:X4}",a);
    Console.WriteLine("Zahl 2: {0:X4}",b);
  }
}
```

Bei einer Eingabe der Zahlen 75 und 1024 würde sich in diesem Beispiel folgende Ausgabe ergeben:

Die Zahlen lauten
Zahl 1: 004B
Zahl 2: 0400

Auch dieses Beispielprogramm finden Sie auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_4\FORMATIERUNG1.

Tabelle 4.4 gibt Ihnen eine Übersicht über die verschiedenen Formatierungszeichen und ihre Bedeutung.

| Zeichen | Formatierung |
|---|---|
| C,c | Währung (engl. *Currency*), formatiert den angegebenen Wert als Preis unter Verwendung der landesspezifischen Einstellungen. |
| D,d | Dezimalzahl (engl. *Decimal*), formatiert einen Gleitkommawert. Die Präzisionszahl gibt die Anzahl der Nachkommastellen an. |
| E,e | Exponential (engl. *Exponential*), wissenschaftliche Notation. Die Präzisionszahl gibt die Nummer der Dezimalstellen an. Bei wissenschaftlicher Notation wird immer mit einer Stelle vor dem Komma gearbeitet. Der Buchstabe „E“ im ausgegebenen Wert steht für „mal 10 hoch“. |
| F,f | Gleitkommazahl (engl. *fixed Point*), formatiert den angegebenen Wert als Zahl mit der durch die Präzisionsangabe festgelegten Anzahl an Nachkommastellen. |
| G,g | Kompaktformat (engl. *General*), formatiert den angegebenen Wert entweder als Gleitkommzahl oder in wissenschaftlicher Notation. Ausschlaggebend ist, welches der Formate die kompaktere Darstellung ermöglicht. |
| N,n | Numerisch (engl. *Number*), formatiert die angegebene Zahl als Gleitkommazahl mit Kommas als Tausender-Trennzeichen. Das Dezimalzeichen ist der Punkt. |
| X,x | Hexadezimal, formatiert den angegebenen Wert als hexadezimale Zahl. Der Präzisionswert gibt die Anzahl der Stellen an. Eine angegebene Zahl im Dezimalformat wird automatisch ins Hexadezimalformat umgewandelt. |

*Tabelle 4.4: Die Formatierungszeichen von C#*

### 4.5.2 Selbst definierte Formate

Sie haben nicht nur die Möglichkeit, die Standardformate zu benutzen. Sie können die Ausgabe auch etwas direkter steuern, indem Sie in einem selbst definierten Format die Anzahl der Stellen und die Art der Ausgabe festlegen. Tabelle 4.5 listet die verwendeten Zeichen auf.

| Zeichen | Verwendung |
|---|---|
| # | Platzhalter für eine führende oder nachfolgende Leerstelle |
| 0 | Platzhalter für eine führende oder nachfolgende 0 |
| . | Der Punkt gibt die Position des Dezimalpunkts an. |
| , | Jedes Komma gibt die Position eines Tausendertrenners an. |
| % | Ermöglicht die Ausgabe als Prozentzahl, wobei die angegebene Zahl mit 100 multipliziert wird |

*Tabelle 4.5: Formatierungszeichen für selbst definierte Formate*

| Zeichen | Verwendung |
|---|---|
| E+0<br>E-0 | Das Auftreten von E+0 oder E-0 nach einer 0 oder nach dem Platzhalter für eine Leerstelle bewirkt die Ausgabe des Werts in wissenschaftlicher Notation. |
| ; | Das Semikolon wirkt als Trenner für Zahlen, die entweder größer, gleich oder kleiner 0 sind. Die erste Formatierungsangabe bezieht sich auf positive Werte, die zweite auf den Wert 0 und die dritte auf negative Werte. Werden nur zwei Sektionen angegeben, gilt die erste Formatierungsangabe sowohl für positive Zahlen als auch für den Wert 0. |
| \ | Der Backslash bewirkt, dass das nachfolgende Zeichen so ausgegeben wird, wie Sie es in den Formatierungsstring schreiben. Es wirkt nicht als Formatierungszeichen. |
| ' | Wollen Sie mehrere Zeichen ausgeben, die nicht als Teil der Formatierung angesehen werden, können Sie diese in einfache Anführungszeichen setzen. |

*Tabelle 4.5: Formatierungszeichen für selbst definierte Formate (Forts.)*

Ein Beispiel soll auch hier verdeutlichen, wie Sie mit diesen Zeichen arbeiten. Wir wollen eine Zahl im Währungsformat ausgeben, wobei wir die deutsche Währung dahinter schreiben. Ist die Zahl negativ, soll sie in Klammern gesetzt werden. Außerdem verwenden wir Tausendertrennzeichen.

```
/* Beispiel Formatierung 3 */
/* Autor:   Frank Eller    */
/* Sprache: C#             */

using System;

public class Beispiel
{
  public static void Main()
  {
    int a = 0;
    Console.Write("Geben Sie eine Zahl ein: ");
    a = Console.ReadLine();
    Console.WriteLine("{0:#,#.00' DM';(#,#.00)' DM'}",a);
  }
}
```

Sie finden das Programm auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_4\FORMATIERUNG3.

## 4.6 Zusammenfassung

In diesem Kapitel haben wir uns mit den verschiedenen Standard-Datentypen beschäftigt, die C# zur Verfügung stellt. Es ging dabei nicht nur darum, welche Datentypen es gibt, sondern auch, wie man damit arbeitet und z.B. Werte von einem in den anderen Datentyp konvertiert. Besonders behandelt haben wir in diesem Zusammenhang den Datentyp `string`, der eine Sonderstellung einnimmt.

Strings sind ein recht universeller Datentyp und werden daher auch sehr häufig benutzt. Deshalb ist es auch sinnvoll, mehr über diesen Datentyp zu erfahren. Mit Hilfe von Strings ist es möglich, Zeichenketten zu verwalten und auch andere Daten zu formatieren.

In diesem Zusammenhang haben wir uns auch nochmals den Platzhaltern zugewendet und verschiedene Möglichkeiten der Formatierung unterschiedlicher Datentypen durchgesprochen.

## 4.7 Kontrollfragen

Auch für dieses Kapitel habe ich wieder einen Satz Fragen zusammengestellt, der das bisher erworbene Wissen ein wenig vertiefen soll. Gehen Sie die Fragen sorgfältig durch, die Antworten finden Sie im letzten Kapitel.

1. Welcher Standard-Datentyp ist für die Verwaltung von 32-Bit-Ganzzahlen zuständig?
2. Was ist der Unterschied zwischen impliziter und expliziter Konvertierung?
3. Wozu dient ein `checked`-Programmblock?
4. Wie wird die explizite Konvertierung auch genannt?
5. Worin besteht der Unterschied zwischen den Methoden `Parse()` und `ToInt32()` bezogen auf die Konvertierung eines Werts vom Typ `string`?
6. Wie viele Bytes belegt ein Buchstabe innerhalb eines Strings?
7. Was wird verändert, wenn das Zeichen @ bei einem `string` verwendet wird?
8. Welche Escape-Sequenz dient dazu, einen Wagenrücklauf durchzuführen ( eine Zeile weiter zu schalten)?
9. Was bewirkt die Methode `Concat()` des Datentyps `string`?
10. Was bewirkt das Zeichen # bei der Formatierung eines String?
11. Wie können mehrere Zeichen innerhalb einer Formatierungssequenz exakt so ausgegeben werden, wie sie geschrieben sind?
12. Was bewirkt die Angabe des Buchstabens `G` im Platzhalter bei der Formatierung einer Zahl, wie z.B. in `{0:G5}`?

## 4.8 Übungen

In diesen Übungen beschäftigen wir uns mit Zahlen und deren Darstellung. Natürlich werden wir das im vorigen Kapitel Gelernte nicht außer Acht lassen.

### Übung 1

Erstellen Sie eine neue Klasse mit zwei Feldern, die `int`-Werte aufnehmen können. Stellen Sie Methoden zur Verfügung, mit denen diese Werte ausgegeben und eingelesen werden können. Standardmäßig soll der Wert der Felder 0 sein.

### Übung 2

Schreiben Sie eine Methode, in der Sie die beiden Werte dividieren. Das Ergebnis soll aber als `double`-Wert zurückgeliefert werden.

### Übung 3

Schreiben Sie eine Methode, die das Gleiche tut, den Wert aber mit drei Nachkommastellen und als `string` zurückliefert. Die vorherige Methode soll weiterhin existieren und verfügbar sein.

### Übung 4

Schreiben Sie eine Methode, die zwei `double`-Werte als Parameter übernimmt, beide miteinander multipliziert, das Ergebnis aber als `int`-Wert zurückliefert. Die Nachkommastellen dürfen einfach abgeschnitten werden.

### Übung 5

Schreiben Sie eine Methode, die zwei als `int` übergebene Parameter dividiert. Das Ergebnis soll als `short`-Wert zurückgeliefert werden. Falls die Konvertierung nach `short` nicht funktioniert, soll das abgefangen werden. Überladen Sie die bestehenden Methoden zum Dividieren der Werte in den Feldern der Klasse.

### Übung 6

Schreiben Sie eine Methode, die zwei `string`-Werte zusammenfügt und das Ergebnis als `string`, rechtsbündig, mit insgesamt 20 Zeichen, zurückliefert. Erstellen Sie für diese Methode eine eigene Klasse und sorgen Sie dafür, dass die Methode immer verfügbar ist.

# 5 Ablaufsteuerung

Ein Programm besteht, wie wir schon gesehen haben, aus diversen Klassen, die miteinander interagieren. Innerhalb der Klassen wird die Funktionalität durch Methoden zur Verfügung gestellt, in denen Anweisungen für die Durchführung der Funktionen zuständig sind. Einige Basisanweisungen haben wir bereits kennen gelernt, allerdings kann man mit diesen Anweisungen noch nicht besonders gut auf die Anforderungen an ein Programm reagieren.

In diesem Kapitel werden wir uns mit dem Programmablauf beschäftigen, mit der Steuerung von Anweisungen. Ein großer Teil eines Programms besteht aus Entscheidungen, die je nach Aktion des Benutzers getroffen werden müssen, und aus sich wiederholenden Programmteilen, die bis zur Erfüllung einer bestimmten Bedingung durchlaufen werden. Kurz gesagt, in diesem Kapitel geht es um Schleifen und Bedingungen. Nebenbei werden die bisher erlangten Kenntnisse über Klassen, Methoden und Namensräume vertieft. Sie werden bald feststellen, dass eine gewisse Routine einkehrt, was die Verwendung dieser Features angeht. Zunächst jedoch müssen wir wieder Basisarbeit leisten und noch ein paar grundlegende Dinge besprechen.

## 5.1 Absolute Sprünge

*goto*

Innerhalb eines Gültigkeitsbereichs (also eines durch geschweifte Klammern eingeklammerten Anweisungsblocks) ist es möglich, einen absoluten Sprung zu einem bestimmten Punkt innerhalb des Blocks durchzuführen. Die entsprechende Anweisung heißt `goto` und benötigt ein so genanntes *Label* als Ziel.

Bei dieser Anweisung scheiden sich allerdings die Geister. Manche Programmierer halten sie für sinnvoll und begründen dies mit der Aussage, dass auch andere Anweisungen im Prinzip nichts anderes seien als absolute Sprünge innerhalb eines Programms. Andere wie-

derum behaupten, **goto** sei eine Anweisung, die nie benötigt werde. In jedem Fall aber ist sie in C# enthalten und bietet tatsächlich die Möglichkeit, aus tief verschachtelten Schleifen (zu denen kommen wir weiter hinten in diesem Kapitel noch) herauszukommen.

Es gibt drei Arten von **goto**-Sprüngen, um zwei davon werden wir uns im Zusammenhang mit der ebenfalls später noch behandelten **switch**-Anweisung noch kümmern. Die Anweisung, die ich hier besprechen will, bezieht sich auf den Sprung zu einem bestimmten Punkt innerhalb des aktuellen Codeblocks, der durch ein Label gekennzeichnet wird.

Die Syntax einer solchen **goto**-Anweisung lautet wie folgt:

***Syntax***

```
goto Labelbezeichner;

  // ... Anweisungen ...

Labelbezeichner:
  //Anweisungen
```

Ein Label als Zielpunkt für den absoluten Sprung wird durch den Labelbezeichner und einen Doppelpunkt definiert. Für den Labelbezeichner gelten die gleichen Regeln wie für andere Bezeichner innerhalb von C#.

Zu beachten ist hierbei, dass die Anweisungen hinter dem Labelbezeichner in jedem Fall ausgeführt werden, wenn das Programm an diese Stelle kommt. D. h. selbst wenn kein Sprung zu einem Label erfolgt, werden die darin enthaltenen Anweisungen ausgeführt. Ein Beispiel soll dies verdeutlichen:

```
/* Beispiel Absolute Sprünge 1 */
/* Autor:   Frank Eller       */
/* Sprache: C#               */

using System;

public class gotoDemo
{
  public void CountToFive(bool toTen)
  {
    int i=0;

    Zaehlen:
      i++;

      if (!toTen && (i==5))
```

```
        goto Fertig;
      if (i<10)
        goto Zaehlen;

    Fertig:
      Console.WriteLine("Zählung bis {0}",i);
      Console.WriteLine("Zählung fertig");
  }
}
```

Das Beispiel entspricht einer Schleife. Über den Parameter `toTen` kann angegeben werden, ob bis zehn gezählt werden soll oder nur bis fünf. In jedem Fall aber wird das Ergebnis der Zählung am Ende der Routine angezeigt. Die bedingte Verzweigung mittels einer `if`-Anweisung werden wir im nächsten Abschnitt eingehender erläutern, für dieses Beispiel ist das Verständnis derselben noch nicht so wichtig. Sie finden ein funktionierendes Programm mit der angegebenen Klasse auf der beiiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_5\SPRÜNGE1.

Die Funktionalität der obigen Methode kann natürlich wesentlich einfacher erreicht werden, nämlich in Form einer „richtigen“ Schleife unter Verwendung der dafür vorgesehenen Konstrukte. Für die Demonstration der `goto`-Anweisung genügt es jedoch. Nun noch ein Beispiel für einen Sprung, der nicht funktioniert:

```
/* Beispiel Absolute Sprünge 2 */
/* Autor:   Frank Eller       */
/* Sprache: C#               */

using System;

public class gotoDemo
{
  public void CountToFive(bool toTen)
  {
    int i=0;

    Zaehlen:
      i++;
      if (!toTen && (i==5))
      {
        goto Fertig;
      }
      if (i<10)
      {
        goto Zaehlen;
      }
    if ((i==10) || (i==5))
```

```
    {
      Fertig:
        Console.WriteLine("Zählung bis {0}",i);
        Console.WriteLine("Zählung fertig");
    }
  }
}
```

Der Sprung kann hier nicht funktionieren, da das Label `Fertig` in einem eigenen Code-Block deklariert ist, somit in einem anderen Gültigkeitsbereich als das `goto`-Statement. Der Sprung zum Label `Zaehlen` funktioniert jedoch, denn dieses Label ist in einem übergeordneten Gültigkeitsbereich deklariert, das `goto`-Statement damit innerhalb des Gültigkeitsbereichs des Labels.

Sie finden auch zu diesem Beispiel den Quellcode auf der CD im Verzeichnis BEISPIELE\KAPITEL_5\SPRÜNGE2.

Absolute Sprünge funktionieren nur innerhalb eines Gültigkeitsbereiches. Ist der Gültigkeitsbereich des `goto`-Statements innerhalb des Gültigkeitsbereichs des Sprungziels deklariert, funktioniert der Sprung. Ist das Label allerdings innerhalb eines Codeblocks deklariert, der zu einer Schleife, einer Verzweigungsanweisung oder gar einer anderen Methode gehört, so funktioniert der Sprung nicht. Anders ausgedrückt: Man kann aus einer Schleife herausspringen, nicht aber hinein.

## 5.2 Bedingungen und Verzweigungen

### 5.2.1 Vergleichs- und logische Operatoren

Bedingungen sind immer Kontrollen auf wahr oder falsch, es ergibt sich also für eine Bedingung stets ein boolescher Wert. C# stellt eine Anzahl Operatoren zur Verfügung, die für die Kontrolle einer Bedingung verwendet werden und einen booleschen Wert zurückliefern. Tabelle 5.1 zeigt die Vergleichsoperatoren in der Übersicht.

| Operator | Bedeutung |
|---|---|
| != | Vergleich auf Ungleichheit |
| > | Vergleich auf größer |
| < | Vergleich auf kleiner |
| >= | Vergleich auf größer oder gleich |
| <= | Vergleich auf kleiner oder gleich |

*Tabelle 5.1: Vergleichsoperatoren*

Es kommt auch häufig vor, dass mehrere Bedingungen zusammen verglichen werden müssen, z.B. wenn zwei verschiedene Bedingungen wahr sein müssen. Um diese in einer Bedingung zusammenfassen zu können, bietet C# auch logische Operatoren an. Sie finden diese in Tabelle 5.2.

| Operator | Bedeutung |
|---|---|
| ! | nicht-Operator (aus wahr wird falsch und umgekehrt) |
| && | und-Verknüpfung (beide Bedingungen müssen wahr sein) |
| \|\| | oder-Verknüpfung (eine der Bedingungen muss wahr sein) |

*Tabelle 5.2: logische Operatoren*

Bedingungen können weiterhin mittels runden Klammern zusammengefasst werden, so dass auch eine große Anzahl Bedingungen kontrolliert werden kann.

Die Bedingungen werden in C# an vielen Stellen benötigt. Die erste und wohl mit am häufigsten eingesetzte Möglichkeit ist die Verzweigung nach bestimmten Gesichtspunkten. Hierzu bietet C# zwei verschiedene Möglichkeiten. Kommen wir zunächst zur `if`-Anweisung, die eine *Wenn-dann*-Verzweigung darstellt.

### 5.2.2 Die if-Anweisung

Eine der am häufigsten benutzten Funktionen einer Programmiersprache ist immer die Verzweigung, die grundsätzlich auf einer Bedingung basiert. Man führt einen bestimmten Programmschritt aus, wenn eine Bedingung wahr ist, und einen anderen, wenn eine Bedingung falsch ist. In C# existiert hierfür die `if`-Anweisung. Ein Ablaufschema zeigt Abbildung 5.1.

Der `else`-Teil der Anweisung ist dabei optional, Sie können darauf verzichten, wenn er nicht zwingend benötigt wird. Die Syntax der `if`-Anweisung sieht wie folgt aus:

***Syntax***

```
if (Bedingung)
{
  //Anweisungen, wenn Bedingung wahr
};
else
{
  //Anweisungen, wenn Bedingung falsch
}
```

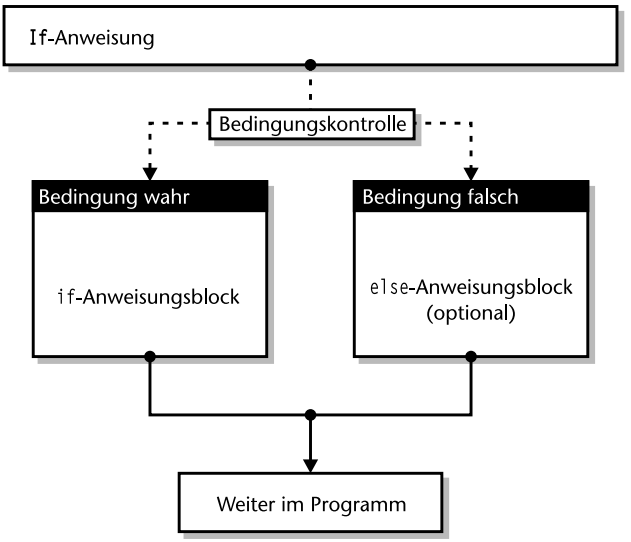


*Abbildung 5.1: Die if-Anweisung bildlich dargestellt*

Die Einfassung der Anweisungen in geschweifte Klammern ist natürlich nur dann notwendig, wenn es sich um mehrere Anweisungen handelt. Bei nur einer Anweisung benötigen Sie keinen Programmblock. Ein Beispiel für die **if**-Anweisung sind zwei Methoden, die entweder die größere oder die kleinere der übergebenen Zahlen zurückliefern, was mittels der **if**-Anweisung problemlos möglich ist. Für dieses Beispiel habe ich die beiden Methoden in einer Klasse zusammengefasst.

```
/* Beispielklasse if-Anweisung 1 */
/* Autor:   Frank Eller          */
/* Sprache: C#                   */

using System;

public class Comparator
{
  public static int IsBigger(int a, int b)
  {
    if (a>b)
      return a;
    else
      return b;
  }

  public static int IsSmaller(int a, int b)
  {
```

```
    if (a<b)
      return a;
    else
      return b;
  }
}
```

Dieses Beispiel zeigt die Verwendung der if-Anweisung recht deutlich. Auch hier beachten Sie bitte, dass C# Groß- und Kleinschreibung berücksichtigt (man kann es gar nicht oft genug betonen). Auf der CD finden Sie natürlich wieder ein Programm zu dieser Beispielklasse, das mit einer entsprechenden Hauptklasse ausgerüstet und funktionsfähig ist. Das Programm finden Sie im Verzeichnis BEISPIELE\KAPITEL_5\IF-ANWEISUNG.

Wenn Sie mehrere Anweisungen zusammenfassen müssen, schließen Sie diese einfach in geschweifte Klammern ein. Damit erzeugen Sie einen Codeblock, der vom Compiler wie eine einzelne Anweisung verstanden wird.

Als Beispiel will ich nun meine Klasse zählen lassen, wie oft die jeweilige Anweisung ausgeführt wird. Ich werde also zwei Variablen zum Zählen hinzufügen und diese jeweils um eins erhöhen, wenn eine Kontrolle durchgeführt wird.

```
/* Beispielklasse if-Anweisung 2 */
/* Autor:   Frank Eller          */
/* Sprache: C#                   */

using System;

public class Comparator
{

  public static int wasSmaller;
  public static int wasBigger;

  public static int IsBigger(int a, int b)
  {
    if (a>b)
    {
      wasBigger++;
      return a;
    }
    else
    {
      wasSmaller++;
```

```
      return b;
    }
  }

  public static int IsSmaller(int a, int b)
  {
    if (a<b)
    {
      wasSmaller++;
      return a;
    }
    else
    {
      wasBigger++;
      return b;
    }
  }
}
```

Die Variable `wasSmaller` wird nun immer um eins erhöht, wenn der erste übergebene Wert kleiner war als der zweite (und zwar unabhängig davon, welche der Funktionen aufgerufen wurde), die Variable `wasBigger` wird dann um eins erhöht, wenn der erste Wert größer war als der zweite.

### 5.2.3 Die switch-Anweisung

`if`-Anweisungen sind eine sehr nützliche Sache. Es gibt allerdings Momente, da ein Programm beim Gebrauch dieser Konstruktion sehr schnell unübersichtlich wird, schlicht, weil zu viele Anweisungen verschachtelt programmiert werden. Ein weiterer Punkt, an dem die `if`-Anweisung undurchsichtig wird, ist die Verzweigung anhand des Werts einer Variablen, d.h. je nachdem, welchen Wert die Variable hat, soll ein anderer Programmteil ausgeführt werden. Für derartige Verzweigungen gibt es eine andere Konstruktion, die speziell für diesen Fall existiert, nämlich die `switch`-Anweisung.

Die `switch`-Anweisung kontrolliert einen Wert und verzweigt dann entsprechend im Programm. Die Funktionsweise können Sie aus dem Diagramm in Abbildung 5.2 ersehen.

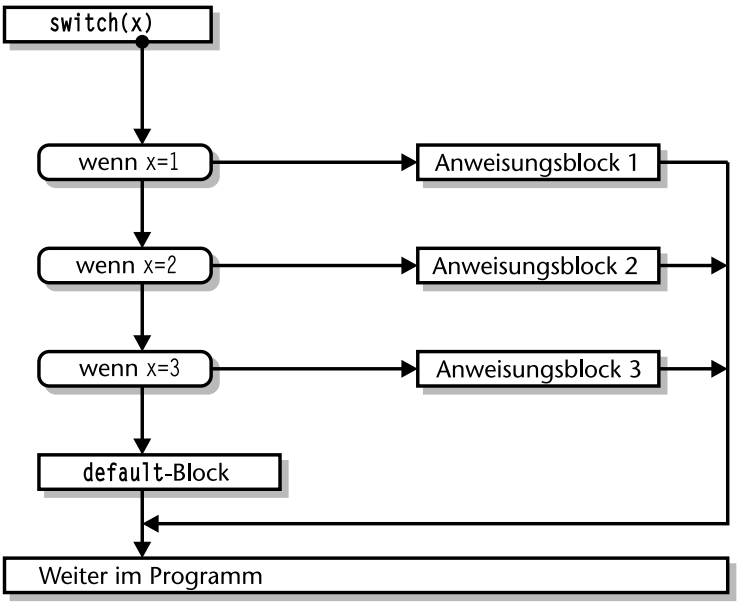


*Abbildung 5.2: Das Ablaufschema der switch-Anweisung*

Die Syntax stellt sich wie folgt dar:

**Syntax**

```
switch (variable)
{
  case 1:
    //Anweisungen ...
    break;

  case 2:
    //Anweisungen ...
    break;

  case 3:
    //Anweisungen ...
    break;

  default:
    //Standard-Anweisungen
}
```

**default**

Die Anweisungen für `default` sind nicht zwingend notwendig. Wird `default` benutzt, dann sind die Anweisungen dahinter der Standard dafür, dass der Wert der Variablen mit keinem der `case`-Statements übereinstimmt. Wenn Sie sicher sind, dass eines der `case`-Statements in jedem Fall angesprungen wird oder wenn für den Fall, dass keines

angesprungen wird, keine Anweisungen notwendig sind, können Sie `default` auch einfach weglassen.

***break***

Wichtig ist, dass die Liste der Anweisungen für jeden `case`-Block mit `break` oder einer anderen Anweisung, die das Verlassen des `switch`-Blocks bewirkt, beendet wird. Diejenigen, die von C++ kommen, werden dies kennen, ist es doch dort ebenfalls so, dass ein so genannter *Fallthrough* durch die einzelnen `case`-Statements ebenfalls durch die `break`-Anweisung verhindert werden muss. Aber es gibt einen gravierenden Unterschied. Während es unter C++ durchaus möglich ist, den Code auch mit fehlenden `break`-Anweisungen zu compilieren, erlaubt der C#-Compiler dies nicht. Die `break`-Anweisungen sind also gefordert, allerdings nur, wenn wirklich Anweisungen vorhanden sind.

***Anweisungsblock für mehrere Übereinstimmungen***

Dieses Verhalten hat den Vorteil, dass Sie nur einen Anweisungsblock benötigen, wenn die gleichen Anweisungen für mehrere Fälle gelten sollen. In diesem Fall können Sie das *Fallthrough*-Verhalten von C# nutzen. Sind keine Anweisungen vorhanden, springt C# das entsprechende `case`-Statement zwar an, fällt aber dann (da ja die Anweisung `break` ebenfalls fehlt) durch zum nächsten `case`-Statement, wenn keine Anweisungen vorhanden sind, wieder durch zum nächsten usw. – bis eben wieder ein Anweisungsblock folgt. Damit ist es leicht, mehreren `case`-Statements einen einzigen Anweisungsblock zuzuweisen, der natürlich seinerseits wieder mit `break` enden muss.

Ein Beispiel für die Verwendung einer `switch`-Anweisung ist die Rückgabe der Anzahl der Tage eines Monats basierend auf der Nummer des Monats im Jahr. Der Januar ist dabei der erste Monat, der Dezember der zwölfte. Eine Klasse mit einer entsprechenden Methode würde wie folgt aussehen:

```
/* Beispielklasse switch-Anweisung */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

using System;

public class Monatskontrolle
{
  public int getDays(int Month)
  {
    switch (Month)
    {
      case 2:
        return 28;
        break;
```

```
      case 4:
      case 6:
      case 9:
      case 11:
        return 30;
        break;

      default:
        return 31;
    }
  }
}
```

Innerhalb der switch-Anweisung wird als Standardwert 31 festgelegt, d.h. wir müssen nur für die Monatszahlen, die sich davon unterscheiden, einen case-Block vorsehen. Der erste Monat ist der Februar (auf Schaltjahre wurde in diesem Beispiel keine Rücksicht genommen), für den der Wert 28 zurückgeliefert wird. Die case-Blöcke für die Monate April, Juni, September und November wurden zusammengefasst. Hier dient das *Fallthrough* bei der switch-Anweisung einem nützlichen Zweck.

Da hinter den Anweisungen das break fehlt, „fällt" der Compiler in einem dieser Fälle durch bis zum case 11, wonach er die Anweisung ausführt und durch break das weitere „Durchfallen" verhindert. Der Compiler fordert das break ja nur, wenn in dem entsprechenden case-Block Anweisungen programmiert wurden. Wir haben also alle case-Statements für die Monate mit 30 Tagen zusammengefasst.

Sie finden das funktionierende Programm auf der CD im Verzeichnis BEISPIELE\KAPITEL_5\SWITCH-ANWEISUNG1.

### 5.2.4 Absolute Sprünge im switch-Block

Die goto-Anweisung haben Sie bereits kennen gelernt. Innerhalb eines switch-Codeblocks können Sie ebenfalls absolute Sprünge durchführen, wobei jedes case automatisch ein Sprungziel definiert. Ebenfalls als Sprungziel definiert ist default, da es sich dabei im Prinzip nur um eine Sonderform des case-Statements handelt. Die beiden Sonderformen für goto sind also

```
goto case;
goto default;
```

Die case-Anweisung muss natürlich vollständig angegeben werden, also mit der entsprechenden Bedingung, z.B.

```
goto case 11;
```

Die goto-Anweisung verlässt den aktuellen Block ebenfalls, in diesem Fall wird allerdings zu einem anderen case-Statement gesprungen. Das birgt die Gefahr, irgendwann in jeder der Anweisungen einen absoluten Sprung zu einem entsprechenden case-Statement programmiert zu haben, d.h. es unmöglich zu machen, die switch-Anweisung zu verlassen. Achten Sie darauf, dass dies nicht passiert, da es den gleichen Effekt hat wie eine ungewollte rekursive Schleife.

### 5.2.5 switch mit Strings

Anders als beispielsweise in C++ muss die Variable, aufgrund der der Vergleich durchgeführt wird, keine numerische bzw. ordinale Variable sein. Es kann sich ebenso um eine string-Variable handeln, was die Möglichkeiten des switch-Statements enorm erweitert. So wäre z.B. eine Passwortkontrolle einfach über eine switch-Anweisung realisierbar, die je nach eingegebenem Namen ein anderes Passwort kontrolliert. Das folgende Beispiel zeigt, wie dies funktioniert.

```
/* Beispiel switch mit Strings */
/* Autor:   Frank Eller        */
/* Sprache: C#                 */

using System;

public class Password
{
  public static string pass1 = "one";
  public static string pass2 = "two";
  public static string pass3 = "three";

  public static bool CheckPassword(string theName,
                                   string pass)
  {
    switch(theName)


    {
      case "Frank Eller":
          return pass.Equals(pass1);
      case "Simone":
          return pass.Equals(pass2);
      case "Klaus Kappler":
          return pass.Equals(pass3);
      default:
```

```
            return false;
        }
    }
}

class TestClass
{
    public static void Main()
    {
        string theName;
        string thePass;

        Console.Write("Geben Sie Ihren Namen ein: ");
        theName = Console.ReadLine();
        Console.Write("Geben Sie das Passwort ein: ");
        thePass = Console.ReadLine();

        if (Password.CheckPassword(theName,thePass))
            Console.WriteLine("Sie sind eingeloggt.");
        else
            Console.WriteLine("Sie sind nicht registriert.");
    }
}
```

Sie finden das Programm auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_5\SWITCH-ANWEISUNG2.

Dieses Beispiel zeigt zwei Dinge. Einmal, dass die Verwendung von Strings bei der **switch**-Anweisung durchaus möglich ist, zum anderen, dass auch die **return**-Anweisung als Ende eines Statements möglich ist. Bei der **return**-Anweisung handelt es sich ja auch um ein Verlassen des **switch**-Blocks, da bekanntlich die gesamte Methode gleich verlassen wird.

In einem **switch**-Block müssen die Anweisungen eines **case**-Statements mit einer Anweisung beendet werden, die das Verlassen des **switch**-Blocks oder den Sprung zu einem anderen **switch**-Block bewirkt. D. h. es sind die Anweisungen **goto**, **break** und **return** erlaubt.

### 5.2.6 Die bedingte Zuweisung

In C# existiert, wie übrigens in C++ auch, eine Form der Zuweisung, die allgemein als *bedingte Zuweisung* bezeichnet wird. Abhängig von einer Bedingung können einer Variablen Werte zugewiesen werden, wobei der erste Wert genommen wird, wenn die Bedingung wahr ist, und der zweite, wenn die Bedingung falsch ist.

Manche Programmierer bezeichnen diese Möglichkeit auch als einen Ersatz zur **if**-Anweisung, das stimmt aber nicht ganz, denn während **if** grundsätzlich eine Verzweigung aufgrund einer Bedingung darstellt, handelt es sich bei der bedingten Zuweisung tatsächlich um eine Zuweisung, es können also keine Methoden aufgerufen werden.

Die Syntax dieser Anweisung liest sich etwas schwierig, da es sich eigentlich nur um zwei Zeichen handelt, nämlich einmal um das Fragezeichen und dann noch um den Doppelpunkt:

***Syntax***

```
Bedingung ? Ausdruck 1 : Ausdruck 2;
```

Mit diesem Wissen können wir unsere Methoden zur Ermittlung des höheren bzw. niedrigeren von zwei Werten umschreiben:

```
/* Beispielklasse bedingte Zuweisung */
/* Autor:   Frank Eller              */
/* Sprache: C#                       */

using System;

public class Comparator
{
  public static int IsBigger(int a, int b)
  {
    return (a>b)?a:b;
  }

  public static int IsSmaller(int a, int b)
  {
    return (a<b)?a:b;
  }
}
```

In diesem Beispiel wird deutlich, dass die bedingte Zuweisung nicht nur mit Variablen, sondern auch mit Rückgabewerten funktioniert. Eigentlich logisch, denn das ist ja im Prinzip auch nur eine Zuweisung.

Die bedingte Zuweisung stellt oftmals eine Erleichterung beim Schreiben dar, man sollte dies allerdings nicht überbewerten. Zwar muss weniger geschrieben werden, der Quelltext kann aber auch sehr schnell sehr unübersichtlich werden, was eine spätere Wartung erschweren kann. Dennoch ist es ein sehr nützliches Feature.

## 5.3 Schleifen

Neben den Verzweigungen sind Schleifen eine weitere Art, den Programmablauf zu steuern. Mit Schleifen können Sie Anweisungsblöcke programmieren, die abhängig von einer Bedingung bzw. für eine festgelegte Anzahl Durchläufe wiederholt werden.

### 5.3.1 Die for-Schleife

Die `for`-Schleife ist die flexibelste Schleifenkonstruktion in C#. Sie wird zwar üblicherweise nur benutzt, wenn die Anzahl der Schleifendurchläufe bekannt ist, das ist aber nicht zwingend notwendig. Sie könnten die `for`-Schleife auch dazu verwenden, die gesamte Funktionalität der anderen Schleifenarten nachzubilden. Abbildung 5.3 zeigt die Funktionsweise der `for`-Schleife.

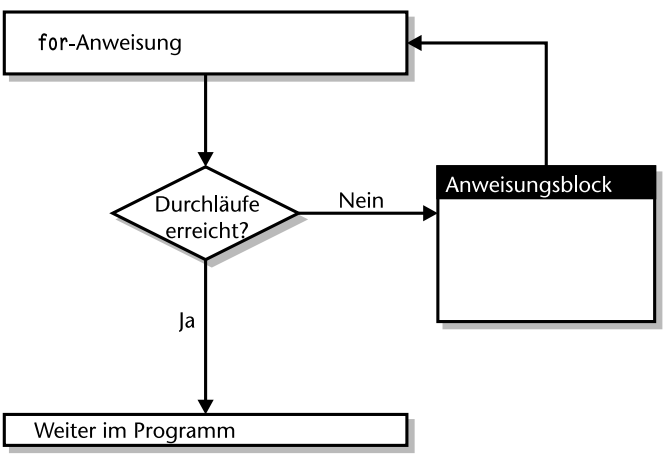


*Abbildung 5.3: Die Funktionsweise der for-Schleife*

Der Kopf der `for`-Schleife ist dreiteilig und besteht aus Initialisierung, einer Bedingungskontrolle und einer Aktualisierung. Üblicherweise handelt es sich bei der Bedingung um eine Kontrolle, ob der Wert der Laufvariable eine bestimmte Größe erreicht hat, und bei der Aktualisierung um die Erhöhung des Werts. Die Syntax der `for`-Schleife stellt sich wie folgt dar:

***Syntax***

```
for (Initialisierung;Bedingung;Aktualisierung)
{
  //Anweisungen
}
```

***Laufvariable***

Die Schleife arbeitet normalerweise mit einer Laufvariablen, die oftmals erst im Kopf der Schleife deklariert und natürlich mit einem Startwert initialisiert wird. Dadurch, dass die Laufvariable erst im Kopf der Schleife deklariert wird, verhält sie sich wie eine lokale Variable, d. h. sie ist dann nur innerhalb des Schleifenkörpers gültig.

***Bedingung***

In der Bedingung wird kontrolliert, ob die Schleife noch einmal durchlaufen werden soll oder nicht. Ebenso wie bei den Verzweigungen wird auch hier ein boolescher Wert verwendet, üblicherweise die Kontrolle der Laufvariablen auf einen bestimmten Wert. Dabei wird die Schleife so lange ausgeführt, wie die Bedingung wahr ist; wird die Bedingung falsch, endet die Schleife und das Programm wird fortgesetzt.

***Aktualisierung***

Die Aktualisierung bezieht sich normalerweise auf die Schleifenvariable, deren Wert dort erhöht oder erniedrigt wird. Bei jedem Schleifendurchlauf wird diese Aktualisierung ausgeführt.

Die `for`-Schleife besteht also im Prinzip aus drei getrennten Anweisungen, die zusammengefasst die Funktionalität ergeben. Eine `for`-Schleife, die die Zahlen von 1 bis 10 auf dem Bildschirm ausgibt, sieht folgendermaßen aus:

```
for (int i=1;i<=10;i++)
  System.Console.WriteLine(i);
```

`i` ist die Laufvariable unserer Schleife. Da wir Variablen überall im Programmtext deklarieren dürfen, deklarieren wir diese Laufvariable direkt im Kopf der Schleife. In der Abbruchbedingung kontrollieren wir, ob `i` kleiner oder gleich dem Wert 10 ist. Ist dies der Fall, dann schreiben wir den Wert von `i` mittels der Methode `WriteLine()`, die Sie ja bereits zur Genüge kennen gelernt haben. Im Aktualisierungsabschnitt erhöhen wir den Wert von `i` dann um eins. Die Schleife wird also automatisch beendet, wenn `i` größer wird als 10.

Ein Beispiel für eine `for`-Schleife im Einsatz ist die Berechnung der Fakultät einer Zahl. Die Fakultät von 1 ist natürlich 1, die Fakultät von 0 ebenfalls. Bei allen anderen Zahlen werden alle Werte bis zur endgültigen Zahl miteinander multipliziert. Als Beispiel die Fakultät von 5 (die 120 beträgt):

```
5! = 1*2*3*4*5 = 120
```

Wenn Sie eine solche Berechnung innerhalb eines Programms durchführen wollten, wäre eine `for`-Schleife eine gute Lösung.

```
/* Beispiel for-Schleife (Fakultätsberechnung) */
/* Autor:   Frank Eller                        */
/* Sprache: C#                                 */

using System;

public class Fakult
{
  public int doFakultaet(int Zahl)
  {
    int fakultaet = 1;

    if (Zahl<2)
      return 1;

    for (int i=1;i<=Zahl;i++)
      fakultaet *= i;

    return fakultaet;
  }
}

public class MainProg
{
  public static int Main(string[] args)
  {
    int theNumber = Int32.Parse(args[0]);
    Fakult myFak = new Fakult();
    Console.WriteLine("Fakultät von {0}:{1}",theNumber,
                      myFak.doFakultaet(theNumber));
    return 0;
  }
}
```

Die Angaben im Kopf der **for**-Schleife sind übrigens optional. Sie müssen weder Initialisierungsteil noch Aktualisierungsteil oder Bedingung angeben, lediglich die Semikola müssen in jedem Fall vorhanden sein. Wenn Sie aber keinerlei Kontrolle oder Aktualisierung im Kopf der **for**-Schleife angeben, müssen Sie selbst dafür sorgen, dass Sie aus der Schleife wieder herauskommen. Die folgende **for**-Schleife würde ebenfalls funktionieren:

```
/* Beispiel for-Schleife (Fakultätsberechnung) */
/* Autor:   Frank Eller                        */
/* Sprache: C#                                 */

using System;

public class Fakult
{
  public int doFakultaet(int Zahl)
  {
    int fakultaet = 1;
    int i = 1;

    if (Zahl<2)
      return 1;

    for (;;)
    {
      fakultaet *= i;
      i++;
      if (i>Zahl)
        break;
    }

    return fakultaet;
  }
}
```

Obwohl im Kopf der Schleife nichts angegeben ist, wird die Schleife durchlaufen. Allerdings wird sie nicht mehr abgebrochen, denn es ist ja keine Bedingung da, die zu kontrollieren wäre. Durch die Anweisung **break** verlassen wir die Schleife, sobald eine bestimmte Bedingung erfüllt ist, in diesem Fall, sobald der Wert der Variable `i` größer ist als der Wert der Variable `Zahl`. Nach dem Verlassen der Schleife geben wir den errechneten Wert zurück.

Beide Programme zur Fakultätsberechnung finden Sie auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_5\FAKULTÄT.

***break***

Die Anweisung **break**, die in dieser Schleife benutzt wurde, dient hier dazu, eben diese Schleife zu verlassen. Genauer ausgedrückt dient **break** dazu, den Programmblock, in dem die Anweisung auftaucht, zu verlassen. Wenn Sie also zwei verschachtelte Schleifen programmiert haben und in der inneren der beiden Schleifen ein **break** programmieren, wird die äußere Schleife dennoch weiter bearbeitet (und die innere damit möglicherweise erneut angestoßen).

Auch wenn Sie **break** in einer Methode verwenden, wird die weitere Behandlung dieser Methode abgebrochen. **break** funktioniert generell mit allen Programmblökken.

***continue***

Das Gegenstück zu **break** ist, zumindest was die Schleifen betrifft, die Anweisung **continue**, die eine Schleife weiterlaufen lässt. Das bedeutet, wenn **continue** aufgerufen wird, beginnt die Schleife von vorne ohne die Anweisungen nach dem **continue**-Aufruf zu bearbeiten. In einer **for**-Schleife bedeutet das, dass die Laufvariable mit jedem **continue** um eins weitergeschaltet wird.

Es wird allerdings nicht nur weitergeschaltet, damit einher geht auch eine erneute Kontrolle der Schleifenbedingung. Sie müssen sich also keine Sorgen machen, es könnte zu einer Endlosschleife kommen; die Kontrolle der Bedingung ist in jedem Fall gewährleistet.

Die Anweisungen **break** und **continue** sind nicht auf die **for**-Schleife oder auf Schleifen allgemein beschränkt. **break** verlässt den aktuellen Programmblock, wenn es aufgerufen wird, unabhängig von der Art der Anweisung. Es ist damit nicht auf Schleifen beschränkt, sondern allgemein gültig. **continue** bezieht sich nur auf Schleifen und startet für die Schleife, in der es programmiert ist, einen neuen Durchlauf.

***lokale Laufvariable***

Als letztes Beispiel zur **for**-Schleife möchte ich beweisen, dass eine im Kopf der Schleife initialisierte Laufvariable wirklich nur im Schleifenblock gültig ist. Das folgende Beispiel ergibt einen Fehler, da bei der zweiten Schleife keine Laufvariable initialisiert ist:

```
/* Beispiel lokale Laufvariable */
/* Autor:   Frank Eller  */
/* Sprache: C#          */

class TestClass
{
  public void ForTest()
  {
    int x = 1;
    for (int i=0;i<10;i++)
    {
      Console.WriteLine("Wert von i: {0}",i);
      x *= i;
      Console.WriteLine("Fakultät von {0}: {1}",i,x);
    }

    //Zweite for-Schleife funktioniert nicht
    for(i=0;i<10;i++)
    {
```

```
      Console.WriteLine("Von {0} wird {1} abgezogen",x,i);
      x -= i;
      Console.WriteLine("x hat den Wert: {0}",x);
    }
  }
}
```

In diesem Beispiel würde die erste **for**-Schleife anstandslos funktionieren, die zweite jedoch nicht. Der Grund hierfür ist die Variable `i`, die in der zweiten Schleife nicht mehr deklariert, sondern nur noch benutzt wurde. Da es sich dabei aber um eine lokale Variable des ersten Schleifenblocks handelt, ist sie dem Compiler nicht bekannt. Die Folge ist ein Fehler. Vielmehr mehrere Fehler, nämlich für jede Verwendung von `i` in der zweiten Schleife einer.

Wird eine Laufvariable im Kopf einer **for**-Schleife deklariert, so ist ihr Gültigkeitsbereich auf den Anweisungsblock der Schleife beschränkt. Es handelt sich dann um eine lokale Schleifenvariable.

### 5.3.2 Die while-Schleife

Eine weitere Form der Schleife, die in Abhängigkeit von einer Bedingung ausgeführt wird, ist die **while**-Schleife. Bei dieser Schleifenart wird der Schleifenkörper so lange durchlaufen, wie die Bedingung wahr ist.

Damit kann hier eine Schleife programmiert werden, die nicht unbedingt durchlaufen wird, denn wenn die Bedingung von Anfang an falsch ist, würde die Schleife sofort übersprungen. Eine solche Schleifenform nennt man auch *abweisende Schleife*.

Die Syntax der **while**-Schleife lautet wie folgt:

***Syntax***

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

Dabei handelt es sich bei der Bedingung natürlich wieder um einen booleschen Wert, der zurückgeliefert werden muss.

Auch für die **while**-Schleife möchte ich Ihnen ein Beispiel liefern, nämlich die Berechnung des ggT, des größten gemeinsamen Teilers, der bei Bruchrechnungen Verwendung findet. Mit Hilfe einer **while**-Schleife kann dieser sehr leicht errechnet werden:

```
/* Beispielklasse while-Schleife (ggT-Berechnung) */
/* Autor:   Frank Eller                           */
/* Sprache: C#                                    */

using System;

public class ggt
{
  public int doGGT(int val1, int val2)
  {
    int helpVal = val1;

    while (helpVal > 1)
    {
      if (((val1 % helpVal)==0)&&
          ((val2 % helpVal)==0))
        break;
      else
        helpVal--;
    }
    return helpVal;
  }
}
```

Das komplette Programm finden Sie auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_5\GGT.

Per Definitionem existiert nicht immer ein größter gemeinsamer Teiler zweier Zahlen. Wenn wir unsere Berechnung durchführen, werden wir feststellen, dass wir dennoch immer auf ein Ergebnis kommen, nämlich im ungünstigsten Fall auf das Ergebnis 1. Das ist auch genau der Fall, für den kein ggT existiert.

Aus diesem Grund benötigen wir auch keine Kontrolle, es wird in jedem Fall ein Wert zurückgeliefert, selbst wenn es nur der Wert 1 ist. Wir wissen aber, wenn 1 zurückgeliefert wird, gibt es keinen größten gemeinsamen Teiler.

Der ggT ist in jedem Fall kleiner als die kleinere der beiden Zahlen. Aus diesem Grund ist es unerheblich, welche der beiden übergebenen Zahlen zur Kontrolle herangezogen wird. In der Variable `helpVal` speichern wir diese Zahl und kontrollieren in der Folge jeweils, ob sich beide übergebenen Zahlen ohne Rest durch `helpVal` teilen lassen. Ist dies der Fall, dann ist `helpVal` auch der ggT, wir können die Methode verlassen. Andernfalls erniedrigen wir `helpVal` um eins und führen die Kontrolle erneut durch. Spätestens wenn `helpVal` den Wert 1 erreicht, wird die Methode beendet.

Wir sehen, dass wir auch in dieser Methode mit dem **break** arbeiten, um die **while**-Schleife vorzeitig zu beenden.

Die **while**-Schleife ist wie gesagt eine abweisende Schleifenform, da es durchaus möglich ist, dass der Code innerhalb des Schleifenblocks nie durchlaufen wird (die Bedingung könnte von Anfang an falsch sein). Wenn Sie sicherstellen möchten, dass der Code mindestens einmal durchlaufen wird, verwenden Sie die **do-while**-Schleife.

### 5.3.3 Die do-while-Schleife

Auch die **do**-Schleife (oder **do-while**-Schleife) ist abhängig von einer Bedingung. Anders als bei der **while**-Schleife wird hier der Code aber mindestens einmal durchlaufen, weil die Bedingung erst am Ende der Schleife kontrolliert wird, weshalb man auch von einer *nicht-abweisenden Schleife* spricht. Die Syntax der **do**-Schleife lautet wie folgt:

***Syntax***

```
do
{
  //Anweisungen
} while (Bedingung);
```

Ein Beispiel für die Verwendung einer **do**-Schleife ist die Berechnung der Quadratwurzel nach Heron. Es handelt sich dabei um ein Annäherungsverfahren, das bereits nach einer kleinen Anzahl an Schritten ein verhältnismäßig genaues Ergebnis liefert.

***Quadratwurzel nach Heron***

Heron ging von folgender Überlegung aus: Wenn ein Quadrat existiert, dessen Fläche meiner Zahl entspricht, muss die Kantenlänge dieses Quadrats zwangsläufig der Wurzel der Zahl entsprechen. Wenn ich also ein Rechteck nehme, bei dem eine Kante die Länge 1 besitzt und die andere Kante die Länge meiner Zahl hat, so hat dieses Rechteck auch die gleiche Fläche. Alles, was nun noch zu tun bleibt, ist, das arithmetische Mittel zweier Kanten zu errechnen und daraus ein neues Rechteck zu bilden, bis es sich um ein Quadrat handelt – die errechnete Kantenlänge ist also dann die gesuchte Wurzel.

Das arithmetische Mittel lässt sich leicht berechnen, es handelt sich um eine einfache mathematische Formel. Unter der Annahme, dass ein Rechteck die Seiten *a* und *b* hat, die neuen Seitenlängen *a'* und *b'* lauten und die Fläche des Rechtecks mit *A* bezeichnet wird, lässt sich das arithmetische Mittel der Seiten *a* und *b* folgendermaßen errechnen:

$$a' = (a+b)/2$$

Die Errechnung der neuen Seite *b'* erfolgt über die Fläche, die uns ja bekannt ist:

*b' = A/a'*

Vorausgesetzt, dass wir eine Genauigkeit festlegen, bei deren Erreichen die Berechnung beendet werden soll, können wir die Wurzel einer Zahl mit einer einfachen **do**-Schleife berechnen. Im Beispiel steht die Konstante `g` für die Genauigkeit, die für unsere Berechnung gelten soll.

```
/* Beispielklasse do-Schleife (Heron) */
/* Autor:   Frank Eller               */
/* Sprache: C#                        */

using System;

class Heron
{
  public double doHeron(double a)
  {
    double A = a;
    double b = 1.0;
    const double g = 0.0004;

    do
    {
      a = (a+b)/2;
      b = A/a;
    }
    while ((a-b)>g);

    return a;
  }
}

class TestClass
{
  public static void Main()
  {
    double x = 0;
    Heron h = new Heron();

    Console.Write("Geben Sie einen Wert ein: ");
    x = Console.ReadLine().ToDouble();
    Console.WriteLine("Wurzel von {0} ist {1}",
                      x,h.doHeron(x));
  }
}
```

Die Quadratwurzelberechnung nach Heron finden Sie auf der CD im Verzeichnis BEISPIELE\KAPITEL_5\HERON. Wenn Sie wollen, können Sie das Programm auch so abändern, dass auch die einzelnen Zwischenschritte der Berechnung angezeigt werden.

Sie sehen in diesem Beispiel, dass C# tatsächlich Groß- und Kleinschreibung unterscheidet. Hier wurden zwei lokale Variablen deklariert, beide vom Typ `double`, nämlich einmal der übergebene Parameter `a` und dann die Variable für die Fläche `A`. Beide Variablen werden vom Compiler getrennt behandelt, weil die eine groß-, die andere kleingeschrieben ist.

Sie sollten allerdings bei der Namensvergabe aufpassen, wenn Sie diese Möglichkeit nutzen. In diesem Fall hat es sich angeboten, da `A` ohnehin in der Mathematik für die Fläche steht und `a` für die Länge einer Kante eines Rechtecks. Hier war also fast keine Verwechslung mehr möglich. Doch schnell hat man sich auf einmal verschrieben. Deshalb nochmals der Appell: Benutzen Sie wenn irgend möglich aussagekräftige Bezeichner für Ihre Variablen und Methoden.

## 5.4 Zusammenfassung

In diesem Kapitel haben Sie verschiedene Möglichkeiten kennen gelernt, den Programmablauf entsprechend Ihrer Vorstellungen zu beeinflussen. Angefangen von der selten benutzten (und von vielen Programmierern als sinnlos angesehenen) goto-Anweisung über die Möglichkeiten der Verzweigung bis hin zu verschiedenen Schleifenkonstruktionen.

## 5.5 Kontrollfragen

Die folgenden Fragen und Übungen sollen wieder der Vertiefung Ihrer Kenntnisse dienen.

1. Wozu dient die `goto`-Anweisung?
2. Welchen Ergebnistyp muss eine Bedingung für eine Verzweigung liefern, wenn die `if`-Anweisung benutzt werden soll?
3. Welcher Datentyp muss für eine `switch`-Anweisung verwendet werden?
4. Wann spricht man von einer nicht-abweisenden Schleife?
5. Wie müsste eine Endlosschleife aussehen, wenn sie mit Hilfe der `for`-Anweisung programmiert w_re?
6. Was bewirkt die Anweisung `break`?

7. Was bewirkt die Anweisung `continue`?
8. Ist die Laufvariable einer `for`-Schleife, wenn sie im Schleifenkopf deklariert wurde, auch f_r den Rest der Methode g_ltig?
9. Wohin kann inerhalb eines `switch`-Anweisungsblocks mittels der `goto`-Anweisung gesprungen werden?
10. Wie kann man innerhalb eines `switch`-Blocks mehreren Bedingungen die gleiche Routine zuweisen?
11. Warum sollte die bedingte Zuweisung nicht für komplizierte Zuweisungen benutzt werden?

## 5.6 Übungen

### Übung 1

Schreiben Sie eine Funktion, die kontrolliert, ob eine übergebene ganze Zahl gerade oder ungerade ist. Ist die Zahl gerade, soll `true` zurückgeliefert werden, ist sie ungerade, `false`.

### Übung 2

Schreiben Sie eine Methode, mit der überprüft werden kann, ob es sich bei einer übergebenen Jahreszahl um ein Schaltjahr handelt oder nicht. Ein Jahr ist dann ein Schaltjahr, wenn es entweder durch 4, aber nicht durch 100 teilbar ist, oder wenn es durch 4, durch 100 und durch 400 teilbar ist. Die Methode soll `true` zurückliefern, wenn es sich um ein Schaltjahr handelt, und `false`, wenn nicht.

### Übung 3

Schreiben Sie eine Methode, die kontrolliert, ob eine Zahl eine Primzahl ist. Der Rückgabewert soll ein boolescher Wert sein.

### Übung4

Schreiben Sie eine Methode, die den größeren zweier übergebener Integer-Werte zurückliefert.

### Übung 5

Schreiben Sie analog zur Methode ggT auch eine Methode kgV, die das kleinste gemeinsame Vielfache errechnet.

### Übung 6

Erweitern Sie das Beispiel „Quadratwurzel nach Heron“ so, dass keine negativen Werte mehr eingegeben werden können. Wird ein negativer Wert eingegeben, so soll der Anwender darüber benachrichtigt werden und eine weitere Eingabemöglichkeit erhalten.

# 6 Operatoren

Operatoren sind ein wichtiger Bestandteil einer Programmiersprache. Mit ihnen führen Sie Berechnungen oder Vergleiche durch, verknüpfen Bedingungen oder auch Zahlenwerte und weisen Variablen Werte zu. Sie haben bereits einige Operatoren kennen gelernt, weil die Verwendung derselben für die Programmierung unverzichtbar ist. Ein Operator, der in diese Gruppe gehört, ist der Zuweisungsoperator =, den man in der Regel anwendet ohne darüber nachzudenken, weil man ihn aus der Mathematik bereits kennt.

Die Art und Weise, wie die Operatoren verwendet werden, bzw. die Rangfolge der Operatoren (z.B. bei mathematischen Operatoren wie den Grundrechenarten) wurde aus der Programmiersprache C++ übernommen.

In diesem Kapitel werden wir näher auf die verschiedenen Arten der Operatoren, ihre Verwendung und ihren Zweck eingehen. Weiterhin werden Sie lernen, wie Sie in eigenen Klassen Operatoren überladen können, um eine neue Funktionalität hinzuzufügen.

## 6.1 Mathematische Operatoren

Ein wichtiger Bestandteil einer jeden Programmiersprache sind die mathematischen Funktionen. In jedem Programm werden normalerweise irgendwelche Berechnungen durchgeführt. C# führt sogar einen ganz neuen Datentyp für finanzmathematische Berechnungen ein, nämlich den Datentyp `decimal`, einen 128 Bit breiten Gleitkommawert mit 28–29 signifikanten Nachkommastellen. Mit C# können Sie also schon sehr genau rechnen.

### 6.1.1 Grundrechenarten

Kommen wir zunächst zu den Operatoren der Grundrechenarten. Im Vergleich zu den Berechnungen, die Sie von Haus aus kennen, müssen Sie sich bei einer Programmiersprache ein wenig umgewöhnen, vor allem was die Symbolik angeht. So steht das Sternchen (*) für eine Multiplikation, dividiert wird mit einem Schrägstrich (/). Auch bei Berechnungen mit Brüchen werden Sie Ihre Denkweise ein wenig anpassen müssen, denn Bruchstriche gibt es in C# nicht. Stattdessen werden der gesamte Ausdruck, der oberhalb des Bruchstrichs steht, und der Ausdruck, der unterhalb des Bruchstrichs steht, in Klammern eingefasst und einfach dividiert. Ein Beispiel:

```
/* Beispielklasse Mathematik 1 */
/* Autor:   Frank Eller        */
/* Sprache: C#                 */

namespace Beispiel
{
  using System;

  public class Rechnen
  {
    public double Bruch(double a, double b, double c)
    {
      return ((a+b)*c)/(c*b);
    }
  }
}
```

Das obige Beispiel berechnet einen Bruch, wobei der Zähler dem Ausdruck `(a+b)*c` entspricht, der Nenner dem Ausdruck `c*b`. Beide Ausdrücke werden in Klammern eingefasst, somit werden sie getrennt berechnet und dann wird dividiert.

Die Operatoren für die Grundrechenarten finden Sie in Tabelle 6.1.

| Operator | Berechnung |
|---|---|
| + | Der Plus-Operator. Es wird eine Addition durchgeführt. |
| - | Der Minus-Operator. Es wird eine Subtraktion durchgeführt. |
| / | Der Divisions-Operator. Es wird eine Division durchgeführt. Der Ergebnistyp richtet sich nach den verwendeten Datentypen und entspricht immer dem genauesten Typ in der Berechnung. |
| * | Der Multiplikations-Operator |

*Tabelle 6.1: Die Grundrechenarten in C#*

| Operator | Berechnung |
|---|---|
| % | Der Modulo-Operator. Dieser Operator dient dazu, bei einer Division nicht den Ergebniswert, sondern den Wert hinter dem Komma zu erfahren. Der Ausdruck 5%2 ergibt damit 5, denn 5 geteilt durch 2 ist 2,5, der Modulo-Operator nimmt aber nur den Teil hinter dem Komma – die 5. |
| ++ | Der Inkrement-Operator. Dieser Operator addiert 1 zum aktuellen Wert. |
| -- | Der Dekrement-Operator. Dieser Operator vermindert den aktuellen Wert um 1 |

*Tabelle 6.1: Die Grundrechenarten in C# (Forts.)*

**Divisionsoperator ( / )**

Eine Sonderstellung bei den Rechenoperatoren nimmt der Divisionsoperator ein. Dieser führt sowohl Divisionen mit ganzzahligen Werten als auch Divisionen mit Gleitkommawerten durch. Dabei richtet sich das Ergebnis immer nach der Genauigkeit des genauesten Werts in der Berechnung. Nehmen wir zunächst an, wir würden mit zwei `int`-Werten arbeiten und würden diese beiden dividieren. Da `int` der genaueste Datentyp der Berechnung ist, ist auch das Ergebnis vom Datentyp `int` und enthält somit keine Nachkommastellen:

```
/* Beispielklasse Mathematik 2 */
/* Autor:   Frank Eller       */
/* Sprache: C#               */

namespace Beispiel
{
  using System;

  public class Rechnen
  {
    public static void Main()
    {
      int a = 5;
      int b = 2;
      Console.WriteLine("Ergebnis von 5/2: {0}.",(a/b));
    }
  }
}
```

Wenn Sie das obige Programm ausführen, werden Sie folgendes Ergebnis erhalten:

**Ergebnis von 5/2: 2.**

Das Programm finden Sie auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_6\MATHE1.

Der Grund ist, dass ein **int**-Wert keine Nachkommastellen besitzen kann, diese also einfach abgeschnitten werden. Anders sieht es aus, wenn einer der Werte ein **double** ist, also eine höhere Genauigkeit besitzt:

```
/* Beispielklasse Mathematik 3 */
/* Autor:   Frank Eller        */
/* Sprache: C#                 */

namespace Beispiel
{
  using System;

  public class Rechnen
  {
    public static void Main()
    {
      double a = 5;
      int b = 2;
      Console.WriteLine("Ergebnis von 5/2: {0}.",(a/b));
    }
  }
}
```

Jetzt lautet das Ergebnis:

**Ergebnis von 5/2: 2,5.**

Dieses Programm finden Sie auf der CD im Verzeichnis BEISPIELE\KAPITEL_6\MATHE2.

Der genaueste Datentyp gibt also auch den Datentyp an, der als Ergebnis verwendet wird. Sehen Sie sich nun folgende Berechnung an:

```
/* Beispielklasse Mathematik 4 */
/* Autor:   Frank Eller        */
/* Sprache: C#                 */

namespace Beispiel
{
  using System;

  public class Rechnen
  {
    public static void Main()
    {
      double erg;
      int a = 5;
```

```
      int b = 2;

      erg = a/b;

      Console.WriteLine("Ergebnis von 5/2: {0}.",erg);
    }
  }
}
```

Auch dieses Programm finden Sie auf der CD, im Verzeichnis BEISPIELE\KAPITEL_6\MATHE3.

Was, glauben Sie, wird als Ergebnis ausgegeben? Normalerweise sollte man denken, dass der Datentyp **double** der genaueste im Ausdruck ist, daher also wie im Beispiel vorher der Wert 2.5 als Ergebnis ausgegeben wird.

Die Ausgabe des Programms lautet jedoch:

Ergebnis von 5/2: 2.

Der Grund ist, dass zwar eine Umwandlung stattfindet, allerdings erst nachdem das Ergebnis berechnet wurde. Der Datentyp **double** ist genauer als der Datentyp **int**, daher ist eine implizite Umwandlung durch den Compiler möglich, die uns allerdings von außen nicht auffällt. Diese Umwandlung wird aber erst dann durchgeführt, wenn die Berechnung erfolgt ist. Für diese gilt aber, dass der genaueste Datentyp der Datentyp **int** ist, was bedeutet, dass das Ergebnis keine Nachkommastellen enthält. Also wird der Wert „2“ in einen **double**-Wert umgewandelt.

Um das korrekte Ergebnis zu erhalten, muss wenigstens einer der Werte der eigentlichen Rechnung ein Gleitkommawert sein, dann funktioniert es. Natürlich muss dann auch der Wert, dem das Ergebnis zugewiesen wird, eine entsprechende Genauigkeit besitzen. Beim folgenden Beispiel würde der Compiler einen Fehler melden, da die Rechnung als Ergebnis einen Wert vom Typ **double** liefert, die Variable, der das Ergebnis zugewiesen wird, aber vom Typ **int** ist.

```
/* Beispielklasse Mathematik 5 */
/* Autor:   Frank Eller        */
/* Sprache: C#                 */

namespace Beispiel
{
  using System;

  public class Rechnen
```

```
  {
    public static void Main()
    {
      int erg;
      double a = 3.5;
      int b = 2;

      erg = a/b;  //Fehler: Keine Konvertierung möglich

      Console.WriteLine("Ergebnis von 5/2: {0}.",erg);
    }
  }
}
```

Sie finden den Quellcode für dieses Beispiel ebenfalls auf der CD, im Verzeichnis BEISPIELE\KAPITEL_6\MATHE4.

### 6.1.2 Zusammengesetzte Rechenoperatoren

Zusätzlich zu diesen Operatoren besitzt C# noch Rechenoperatoren, bei denen die Zuweisung und die Berechnung zusammengefasst wurden. Sie können also mit einer einzigen Anweisung gleichzeitig rechnen und zuweisen. Diese Anweisungen, die Sie in Tabelle 6.2 finden, werden gebildet, indem Rechenoperator und Gleichheitszeichen zusammengesetzt werden:

| Operator | Bedeutung |
|---|---|
| += | Dieser Operator führt eine Addition mit gleichzeitiger Zuweisung durch. `x +=y` entspricht `x = x+y`. |
| -= | Dieser Operator führt eine Subtraktion mit gleichzeitiger Zuweisung durch. `x -= y` entspricht `x = x-y`. |
| *= | Dieser Operator führt eine Multiplikation mit gleichzeitiger Zuweisung durch. `x *= y` entspricht `x = x*y`. |
| /= | Dieser Operator führt eine Division mit gleichzeitiger Zuweisung durch. `x /= y` entspricht `x = x/y`. |
| %= | Dieser Operator führt die Modulo-Operation mit gleichzeitiger Zuweisung durch. `x %= y` entspricht `x = x%y`. |

*Tabelle 6.2: Zusammengesetzte Rechenoperatoren in C#*

Mit Hilfe dieser Operatoren können einfache Berechnungen natürlich auch einfacher geschrieben werden. Ob sie dadurch übersichtlicher werden, muss von Fall zu Fall gesehen werden. Bei der Behandlung der Schleifen haben wir einen solchen Rechenoperator bereits kennen gelernt, nämlich bei der Berechnung der Fakultät einer Zahl, wo wir die Multiplikation und die Zuweisung kombiniert hatten.

Als Beispiel für die Berechnung mit Hilfe der zusammengesetzten Rechenoperatoren möchte ich noch die Berechnung der Quersumme einer Zahl anführen. Sie wissen, dass die Quersumme einer Zahl aus der Summe aller Ziffern dieser Zahl besteht. Damit können wir mit folgendem Beispiel die Quersumme berechnen:

```
/* Beispiel Mathematik (Quersumme) */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

namespace Quersumme
{
  using System;

  public class Quersumme
  {
    public int DoQuersumme(int theValue)
    {
      int erg = 0;

      do
      {
        erg +=(theValue%10);
        theValue /= 10;
      } while (theValue>0);

      return erg;
    }
  }

  public class MainProg
  {
    public static int Main(string[] args)
    {
      Quersumme myQSum = new Quersumme();

      Console.Write("Bitte geben Sie eine Zahl ein: ");
      int myValue = Console.ReadLine().ToInt32();

      int erg = myQSum.DoQuersumme(myValue);

      Console.WriteLine("Die Quersumme von {0} beträgt {1}",
                        myValue,erg);

      return 0;
    }
  }
}
```

Die eigentliche Berechnung der Quersumme erfolgt, indem wir von dem ursprünglichen Wert immer ein Zehntel „abknabbern“, also immer eine Ziffer, und diese dann dem Ergebnis hinzuaddieren. Die Zeile

```
erg += (theValue % 10);
```

knabbert den Wert ab und addiert die Ziffer zum Ergebnis, die Zeile

```
theValue /= 10;
```

dividiert dann noch den Wert, der ja unverändert geblieben ist, durch 10. Wenn bei dieser Division der Wert 0 erreicht ist, also der letzte Wert addiert worden ist, wird die Schleife für die Berechnung abgebrochen und das Ergebnis an die aufrufende Methode, in diesem Fall die Methode `Main()`, zurückgeliefert.

Das gesamte Programm finden Sie natürlich wieder auf der beiliegenden CD, im Verzeichnis BEISPIELE\KAPITEL_6\QUERSUMME.

### 6.1.3 Die Klasse Math

In der `Math`-Klasse sind einige erweiterte Rechenfunktionen implementiert, alle als statische Methoden, so dass Sie sofort darauf zugreifen können, ohne eine Instanz der Klasse erzeugen zu müssen. `Math` ist im Namensraum `System` deklariert, den Sie ohnehin in Ihrem Programm eingebunden haben, so können Sie die enthaltenen Funktionen direkt nutzen.

`Math` liefert auch zwei statische Felder, die oftmals verwendet werden, nämlich den natürlichen Exponenten `e` und die Zahl π. Einige der wichtigsten Methoden der Klasse `Math` finden Sie in Tabelle 6.3. Es handelt sich dabei ausnahmslos um statische Methoden, die Sie also auf alle Werte anwenden können.

| Methode | Funktion |
|---|---|
| Abs | Liefert den absoluten Wert einer Zahl zurück (mathematisch: den Betrag eines Wertes). |
| Acos | Liefert den Arcuscosinus eines Werts zurück. Der Wert wird im Rad-Format zurückgeliefert. |
| Asin | Liefert den Arcussinus eines Werts zurück. Der Wert wird im Bogenmaß zurückgeliefert. |
| Atan | Liefert den Arcustangens eines Werts zurück. Der Wert wird im Bogenmaß zurückgeliefert. |
| Ceil | Rundet eine Zahl zur nächsthöheren ganzen Zahl auf. |
| Cos | Liefert den Cosinus eines Winkels zurück. Der Wert des Winkels wird im Bogenmaß angegeben. |

*Tabelle 6.3: Die wichtigsten Methoden von Math*

| Methode | Funktion |
|---|---|
| Exp | Liefert $e^x$ zurück, wobei x die angegebene Zahl ist. |
| Floor | Rundet eine Zahl zur nächstkleineren ganzen Zahl ab. |
| Log10 | Liefert den Logarithmus zur Basis 10 eines Wertes zurück. |
| Max | Liefert die größere zweier übergebener Zahlen zurück. |
| Min | Liefert die kleinere zweier übergebener Zahlen zurück. |
| Pow | Setzt eine Zahl zu einer anderen Zahl in die Potenz. |
| Round | Rundet einen Wert mathematisch. |
| Sin | Liefert den Sinus eines Winkels zurück. Der Winkel wird im Bogenmaß angegeben. |
| Sqrt | Liefert die Quadratwurzel einer Zahl zurück. |
| Tan | Liefert den Tangens eines Winkels zurück. Der Winkel wird im Bogenmaß angegeben. |

*Tabelle 6.3: Die wichtigsten Methoden von Math (Forts.)*

Wie Sie sehen, arbeiten die Winkelfunktionen allesamt im Bogenmaß, nicht wie bei uns gewohnt im Gradmaß. Es existiert auch keine Funktion, die uns das Bogenmaß ins Gradmaß umwandelt oder umgekehrt. Daher müssen wir uns eine solche Funktion selbst schreiben.

Die folgende Klasse enthält zwei statische Methoden, die diese Umwandlung für uns vornehmen. Wir benötigen deshalb beide, weil die Arcussinus-, Arcuscosinus- und Arcustangens-Funktionen den Winkel ebenfalls im Bogenmaß zurückliefern, wir daher die Umwandlung nach beiden Seiten zur Verfügung stellen müssen. Die Methoden sind aber nicht weiter schwer verständlich, immerhin handelt es sich nur um eine einfache Umsetzung.

```
/* Beispielklasse Mathematik (Winkelkonvertierung) */
/* Autor:   Frank Eller                            */
/* Sprache: C#                                     */

using System;

public class Converter
{
  public static double DegToRad(double a)
  {
    return (a/180*Math.PI);
  }

  public static double RadToDeg(double a)
  {
    return (a*180/Math.PI);
  }
}
```

Wenn Sie nun eine der Winkelfunktionen aufrufen möchten, müssen Sie eben die entsprechende Umwandlungsfunktion noch dazwischenschalten. So würde der Aufruf für die Berechnung des Sinus von 45° aussehen:

```
x = Math.Sin(Converter.DegToRad(45));
```

Die Konverterklasse finden Sie auf der CD im Verzeichnis BEISPIELE\KAPITEL_6\WINKELKONVERTER.

## 6.2 Logische Operatoren

Ein Computer kann bekanntlich nur mit zwei Zuständen arbeiten, nämlich 0 (kein Strom da) und 1 (Strom da). Diese Einheiten, die nur zwei Zustände annehmen können, nennt man Bits. Und statt mit Dezimalzahlen können Sie auch mit einzelnen Bits arbeiten, sie miteinander verknüpfen, vergleichen oder verschieben. In diesem Abschnitt werden wir auf die einzelnen Möglichkeiten ein wenig eingehen. Bei der Programmierung herkömmlicher Anwendungen werden Sie allerdings relativ selten damit zu tun bekommen.

### 6.2.1 Vergleichsoperatoren

Zunächst müssen wir wissen, wie wir mit Hilfe von Operatoren einen Vergleich durchführen können. Sie haben Vergleichsoperatoren, z.B. den Operator == für die Kontrolle auf Gleichheit zweier Werte, bereits kennen gelernt. Tabelle 6.4 listet alle Vergleichsopatoren von C# auf.

| Operator | Bedeutung |
|---|---|
| ! | Negation. Aus `true` wird `false` und umgekehrt. |
| == | Kontrolle auf Gleichheit. Überprüft, ob die Werte links und rechts des Operators gleich sind. |
| != | Kontrolle auf Ungleichheit. Überprüft, ob die Werte links und rechts des Operators ungleich sind. |
| > | Vergleich auf größer. Überprüft, ob der Wert links des Operators größer ist als der rechte Wert. |
| < | Vergleich auf kleiner. Überprüft, ob der Wert links des Operators kleiner ist als der rechte Wert. |
| >= | Vergleich auf größer oder gleich. Überprüft, ob der Wert links des Operators größer oder gleich dem rechten ist. |
| <= | Vergleich auf kleiner oder gleich. Überprüft, ob der Wert links des Operators kleiner oder gleich dem rechten ist. |

*Tabelle 6.4: Vergleichsoperatoren in C#*

Alle Vergleichsoperatoren liefern einen booleschen Wert zurück, der entsprechend des Vergleichs entweder `true` oder `false` ist. Damit können diese Operatoren z.B. in Schleifen und Verzweigungen verwendet werden.

### 6.2.2 Verknüpfungsoperatoren

Ebenfalls kennen gelernt haben wir bereits Operatoren zum Verknüpfen von Bedingungen oder booleschen Werten. Tabelle 6.5 listet die logischen Verknüpfungen auf.

| Operator | Bedeutung |
|---|---|
| && | Logische und-Verknüpfung. Im Falle zweier boolescher Werte a und b wird `true` zurückgeliefert, wenn a *und* b `true` sind. Aus Gründen der Optimierung wird der zweite Wert nur dann kontrolliert, wenn der erste `true` ist, da ansonsten der gesamte Ausdruck ohnehin nicht mehr `true` werden kann. |
| \|\| | Logische oder-Verknüpfung. Im Falle zweier boolescher Werte a und b liefert a\|\|b `true` zurück, wenn *entweder* a *oder* b `true` sind. Dabei wird - wiederum aus Gründen der Optimierung b nur dann kontrolliert, wenn a `false` ist, da ansonsten der gesamte Ausdruck ohnehin `true` ergeben würde. |

*Tabelle 6.5: Logische Verknüpfungsoperatoren*

Sie müssen bei der Verwendung der logischen Operatoren stets darauf achten, wirklich nur boolesche Werte zu vergleichen. So liefern z.B. die Vergleichsoperatoren zwar einen booleschen Wert zurück, die folgende Verknüpfung würde aber dennoch einen Fehler liefern.

```
/* Beispiel logische Verknüpfung 1 */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

class TestClass
{
  public static void Main()
  {
    int a;
    int b;
    int c;

    a = Int32.Parse(Console.ReadLine());
    b = Int32.Parse(Console.ReadLine());
    c = Int32.Parse(Console.ReadLine());
```

```
    if (a<b&&a<c)
      Console.WriteLine("a ist am kleinsten");
  }
}
```

Für einen Menschen ist vollkommen klar, was mit dem obigen Ausdruck gemeint ist und wie die Verknüpfung funktionieren soll, der Computer aber kann nur nach rein logischen Richtlinien vorgehen. Damit sieht die obige Verknüpfung für den Computer folgendermaßen aus (die Zusammenhänge sind mit Klammern gekennzeichnet):

```
if (a<(b&&a)<c)
```

Diese Art der Verknüpfung ist aber nicht zulässig und führt somit zu einem Fehler. Im Falle der Verknüpfung zweier Vergleiche müssen Sie also die einzelnen Vergleiche in Klammern setzen, um dem Computer die Zusammenhänge korrekt darzustellen. Das folgende Beispiel funktioniert.

```
/* Beispiel logische Verknüpfung 2 */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

class TestClass
{
  public static void Main()
  {
    int a;
    int b;
    int c;

    a = Int32.Parse(Console.ReadLine());
    b = Int32.Parse(Console.ReadLine());
    c = Int32.Parse(Console.ReadLine());

    if ((a<b)&&(a<c))
      Console.WriteLine("a ist am kleinsten");
  }
}
```

### 6.2.3 Bitweise Operatoren

Wie bereits angesprochen arbeiten Computer mit Bits. Damit können Sie auch innerhalb Ihrer eigenen Programme die einzelnen Bits vergleichen bzw. manipulieren. Die bitweisen Operatoren finden Sie in Tabelle 6.6.

| Operator | Bedeutung |
|---|---|
| & | Dieser Operator bewirkt eine bitweise und-Verknüpfung zweier Werte. |
| \| | Dieser Operator bewirkt eine bitweise oder-Verknüpfung zweier Werte. |
| ^ | Dieser Operator bewirkt eine bitweise exklusiv-oder-Verknüpfung zweier Werte. |
| >> | Dieser Operator bewirkt ein Verschieben aller Bits eines Werts um eine Stelle nach rechts. |
| << | Dieser Operator bewirkt ein Verschieben aller Bits eines Werts um eine Stelle nach links. |

*Tabelle 6.6: Die bitweisen Operatoren von C#*

Bitweise Operatoren werden üblicherweise verwendet, um Werte zu maskieren oder wenn Sie einen Wert eben Bit für Bit auswerten wollen. Nehmen wir einfach einmal an, Sie wollten Optionen für ein Programm in einem 32-Bit-Integerwert speichern. Durch die Maskierung (mittels einer bitweisen und-Verknüpfung) kann dann jedes einzelne Bit kontrolliert werden – aus einem 32-Bit-Integerwert wird eine Reihe boolescher Werte, die als Optionen benutzt werden können. Das folgende Beispiel zeigt wie:

```
/* Beispiel bitweise Operatoren */
/* Autor:   Frank Eller         */
/* Sprache: C#                  */

using System;

public class cBoolOptions
{
  int Options;

  public cBoolOptions()
  {
    this.Options = 0;
  }

  public cBoolOptions(int Options)

  {
    this.Options = Options;
  }
```

```
  public bool CheckOption(byte optNr)
  {
    int i = (int)(Math.Pow(2,optNr-1));
    return ((Options&i)==i);
  }

  public void setOption(byte optNr)
  {
    if (!CheckOption(optNr))
      Options += (int)(Math.Pow(2,optNr-1));
  }

  public void delOption(byte optNr)
  {
    if (CheckOption(optNr))
      Options -= (int)(Math.Pow(2,optNr-1));
  }
}
```

Bei Anwendung dieser Klasse werden die Optionen, die gesetzt sind, zunächst durch einen Initialwert vorgegeben. Sinnvollerweise sollte es sich dabei um den Wert 0 handeln, da ansonsten etwas Rechnerei angesagt ist. Deshalb initialisiert der Standard-Konstruktor die Klasse auch mit dem Wert 0, so muss man diesen Wert nicht immer angeben.

Wird eine Option gesetzt oder gelöscht, so kontrollieren wir erst den Status des entsprechenden Bit. Über eine Verknüpfung mit dem übergebenen Wert finden wir heraus, ob die gewünschte Option gesetzt ist oder nicht. Dazu rechnen wir das Bit in einen Dezimalwert um.

***Werte der Bits***

Bei binärer Rechnung entspricht das erste Bit dem Wert $2^0$, das zweite Bit dem Wert $2^1$, das dritte Bit dem Wert $2^2$ usw. Damit können wir mittels der Methode `Pow()` exakt das gewünschte Bit in unserer Hilfsvariablen `i` setzen und die beiden Werte – `i` und `Options` – verknüpfen. Das Casting ist notwendig, weil die Methode `Pow()` das Ergebnis als `double`-Wert zurückliefert, wir aber einen `int`-Wert benötigen.

Bei unserer Kontrolle muss das Ergebnis der Verknüpfung gleich dem Wert sein, mit dem wir kontrollieren. Wenn dem so ist, ist das entsprechende Bit gesetzt, wenn nicht, ist es nicht gesetzt. Der eigentliche Wert der Variable `Options` ist uninteressant, da wir ja bitweise kontrollieren und die Methode `CheckOption` exakt den Status des gewünschten Bit zurückliefert.

Wenn wir nun ein Bit setzen wollen, addieren wir einfach den dem Bit entsprechenden Wert der Variable Options hinzu – das Bit wird 1. Wollen wir eine Option löschen, dann ziehen wir den entsprechenden Wert ab – das Bit wird 0. Da wir aber nichts hinzuaddieren dürfen, wenn das gewünschte Bit bereits gesetzt ist, und nichts abziehen, wenn es nicht gesetzt ist, kontrollieren wir den Status vorher.

Die komplette Klasse incl. einem kompletten Unterbau zum Testen finden Sie auf der CD im Verzeichnis BEISPIELE\KAPITEL_6\BOOLOPTIONS.

### 6.2.4 Verschieben von Bits

Die Operatoren zum Verschieben einzelner Bits eines Werts sind ebenfalls interessant. So könnten Sie z.B. durch mehrfaches Verschieben von Bits einen Dezimalwert in einen Binärwert umrechnen und dann als String ausgeben. Wir verschieben dabei die Bits eines Werts nach einer Richtung, bis wir die gesamte Anzahl Bits verschoben haben. Ein Bit fällt dabei immer heraus, die nächsten rücken nach und bei positiven Werten wird von der anderen Seite mit einer 0 aufgefüllt. Diesen Umstand können wir uns zu Nutze machen.

```
/* Beispiel Bitverschiebung (Integer->Binär) */
/* Autor:   Frank Eller                      */
/* Sprache: C#                               */


public class TestClass
{
  public static string IntToBin(int x)
  {
    //x ist ein 32-Bit Integer-Wert

    string theResult = "";

    for (int i=1;i<=32;i++)
    {
      if ((x&1)==1)
        theResult = "1"+theResult;
      else
        theResult = "0"+theResult;
        x = x >> 1;
    }
    return (theResult);
  }
```

```
  public static void Main()
  {
    int x = 0;

    Console.Write("Geben Sie eine Zahl ein:");
    x = Console.ReadLine().ToInt32();

    Console.WriteLine("Binär: {0}",IntToBin(x));
  }
}
```

Dieses Beispielprogramm finden Sie auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_6\BITVERSCHIEBUNG.

Im Beispiel übergeben wir der Methode `IntToBin()` einen 32-Bit-Integerwert, den wir nun Stück für Stück auswerten, und zwar bitweise. Dazu benutzen wir wieder den Verknüpfungsoperator `&`, kontrollieren allerdings immer nur das erste Bit. Ist dieses 1, fügen wir dem Ergebnis-String eine 1 hinzu, andernfalls eine 0. Dann verschieben wir alle Bits des übergebenen Wertes um eine Stelle nach rechts. Wenn wir 32-mal verschoben haben, sind wir fertig und können das Ergebnis zurückliefern.

Natürlich funktioniert hier auch wieder ein zusammengesetzter Operator. Statt der Zuweisung

```
x = x >> 1;
```

könnte man auch schreiben

```
x >>= 1;
```

***Auffüllen***

Wenn die Bits eines Werts verschoben werden, geht natürlich immer ein Bit verloren, da es sozusagen aus dem Wert herausfällt. Wenn es sich um eine Verschiebung nach rechts handelt, ist es das am weitesten rechts platzierte, bei einer Verschiebung nach links das am weitesten links platzierte. Gleichzeitig wird aber auf der anderen Seite eine Stelle frei, die es aufzufüllen gilt.

Dabei ist es nicht so, dass das herausgefallene Bit auf der anderen Seite wieder eingeschoben wird. In diesem Fall würde sich bei einer Verschiebung um 32 Stellen wieder der gleiche Wert ergeben. Stattdessen wird aber bei positiven Zahlen mit 0 aufgefüllt, bei negativen Zahlen mit 1.

Das Auffüllen der Bits richtet sich nach dem Vorzeichen des Werts. Handelt es sich um einen positiven Wert, dann wird mit 0 aufgefüllt, bei einem negativen Wert mit 1.

## 6.3 Zusammenfassung

In diesem Kapitel ging es um die Operatoren, die C# bereitstellt. Wie jede Programmiersprache kann auch C# ziemlich gut rechnen, d.h. es werden umfassende Rechenoperatoren zur Verfügung gestellt und auch Operatoren zur Manipulation bzw. Kontrolle von Werten fehlen nicht. Sicherlich werden Sie sich schnell an die gebräuchlichsten Operatoren gewöhnen und diese in Ihren Programmen einsetzen.

## 6.4 Kontrollfragen

Wiederum sollen einige Fragen dabei helfen, das Gelernte zu vertiefen, und Ihnen Sicherheit beim Umgang mit den verschiedenen Operatoren zu geben.

1. Welchem Rechenoperator kommt in C# eine besondere Bedeutung zu?
2. In welcher Klasse sind viele mathematische Funktionen enthalten?
3. Welche statische Methode dient der Berechnung der Quadratwurzel?
4. Warum muss man beim Rechnen mit den Winkelfunktionen von C# etwas aufpassen?
5. Vergleichsoperatoren liefern immer einen Wert zurück. Welchen Datentyp hat dieser Wert?
6. Was ist der Unterschied zwischen den Operatoren `&&` und `&`?
7. Mit welchem Wert wird beim Verschieben einzelner Bits einer negativen Zahl aufgefüllt?
8. Wie kann man herausfinden, ob das vierte Bit einer Zahl gesetzt ist?

# 7 Datentypen

Alle Datentypen, die wir bisher kennen gelernt haben, waren Standardtypen, die von der Laufzeitumgebung ohnehin zur Verfügung gestellt wurden. Manchmal reichen diese Datentypen aber nicht aus, wohl von den Werten her, aber nicht von der Funktionalität. Deshalb gibt es noch weitere Arten von Datentypen, die Sie selbst definieren und in Ihren Programmen verwenden können – und zwar wie die Standardtypen auch. In diesem Kapitel werden diese Datentypen vorgestellt, es geht um Arrays, Structs und Aufzählungstypen.

## 7.1 Arrays

Bisher haben wir für jeden Wert, den wir in unseren kleinen Beispielprogrammen benutzt haben, eine eigene Variable deklariert. Bisher waren die Programme auch nicht so umfangreich, dass dies ein Problem dargestellt hätte. Doch stellen Sie sich nun einmal vor, Sie hätten ein Programm, bei dem zehn Zahlen des gleichen Datentyps eingelesen und sortiert werden müssten. Bei der herkömmlichen Deklaration würde das dann so aussehen:

```
int i1,i2,i3,i4,i5,i6,i7,i8,i9,i10;
```

Das wäre ja noch ok. Wie sieht es aber bei der Zuweisung und dem Vergleich der Variablen bzw. beim Sortieren aus? Da ist es dann nicht mehr ganz so einfach, denn jetzt müssen Sie alle Variablen miteinander vergleichen, Daten austauschen, wieder vergleichen, bis alles in Ordnung ist ... das geht in C# auch einfacher.

### 7.1.1 Eindimensionale Arrays

*Deklaration*

Sie können in C# so genannte Arrays, Datenfelder, deklarieren. Damit tragen alle Elemente dieses Datenfelds den gleichen Namen, wer-

den aber über einen Index unterschieden. Im folgenden Array ist die gleiche Anzahl Variablen deklariert wie in der obigen Anweisung:

```
int[] i = new int[10];
```

Die Deklaration mittels **new** ist in diesem Fall notwendig, weil auch ein Array ein Referenztyp ist. Wir müssen also für jedes Array eine Instanz erstellen.

***Zugriff auf Arrays***

Während Sie nun bei der ersten Deklaration die einzelnen Variablen direkt über ihren Namen ansprechen können (bzw. müssen), haben Sie bei der zweiten Variante die Möglichkeit, auf die einzelnen Variablen über einen Index zuzugreifen. Der Index wird in eckigen Klammern direkt hinter den Bezeichner geschrieben:

```
i[2] = 15;
```

Ein Array in C# beginnt immer mit dem Index 0. Das bedeutet, in der obigen Deklaration eines Array mit 10 Werten haben die Indizes die Werte 0 bis 9, was insgesamt 10 Werten entspricht.

Der Vorteil, der sich aus der Verwendung eines Array ergibt, liegt auf der Hand: Sie können Schleifen bzw. die Laufvariable einer Schleife benutzen, um mit den Daten zu arbeiten. Als Beispiel hier zunächst der Vergleich der herkömmlichen Deklaration mit einem Array beim Einlesen der Werte:

```
/* Beispiel Einlesen mehrerer Werte 1 */
/* Autor:   Frank Eller               */
/* Sprache: C#                        */

using System;

public class Einlesen1
{
  public static void Main()
  {
    int i1,i2,i3,i4,i5,i6,i7,i8,i9,i0;

    //Einlesen
    i1 = Console.ReadLine().ToInt32();
    i2 = Console.ReadLine().ToInt32();
    i3 = Console.ReadLine().ToInt32();
    i4 = Console.ReadLine().ToInt32();
    i5 = Console.ReadLine().ToInt32();
    i6 = Console.ReadLine().ToInt32();
    i7 = Console.ReadLine().ToInt32();
    i8 = Console.ReadLine().ToInt32();
```

```
    i9 = Console.ReadLine().ToInt32();
    i0 = Console.ReadLine().ToInt32();

    //Weitere Anweisungen ...
  }
}
```

Und nun im Vergleich dazu das Einlesen der gleichen Anzahl Daten in ein Array mit zehn Elementen:

```
/* Beispiel Einlesen mehrerer Werte 2 */
/* Autor:   Frank Eller               */
/* Sprache: C#                        */

using System;

public class Einlesen1
{
  public static void Main()
  {
    int[] i = new int[10];

    //Einlesen
    for (int u=0;u<=9;u++)
      i[u] = Console.ReadLine().ToInt32();

    //Weitere Anweisungen ...
  }
}
```

Sie sehen, dass es doch ein erheblicher Unterschied ist, allein schon was die Schreibarbeit beim Einlesen betrifft. Stellen Sie sich mal vor, Sie müssten mit 100 Werten arbeiten und hätten keine Arrays zur Verfügung ...

### Beispiel: Eine Sortier-Routine

Ein gutes Beispiel für die Verwendung von Arrays ist eine Sortier-Routine mit Zahlen. Die Routine soll mit einer beliebigen Anzahl von Zahlen arbeiten. Wir werden eine eigene Klasse für die Sortierroutine erstellen und die Sortierung nach einem zwar stellenweise langsamen, dafür aber leicht verständlichen Algorithmus durchführen.

**Bubblesort**

Der Algorithmus, den wir verwenden wollen, ist auch als *Bubblesort*-Algorithmus bekannt. Wir haben unser Array mit Werten, das wir nun von vorne nach hinten durchgehen. Finden wir zwei benachbarte Werte, die sich nicht in der richtigen Reihenfolge befinden, ver-

tauschen wir diese und merken uns, dass ein Tausch stattgefunden hat.

Am Ende des Durchgangs wird dies kontrolliert und entsprechend weitergemacht. Falls kein Tausch stattgefunden hat, können wir sicher sein, dass sich alle Elemente in der richtigen Reihenfolge befinden, wir können die Sortierroutine verlassen und das Ergebnis ausgeben. Hat ein Tausch stattgefunden, gehen wir unser Array eben nochmals durch, bis alle Elemente sortiert sind. Abbildung 7.1 zeigt, wie das Sortieren funktioniert.

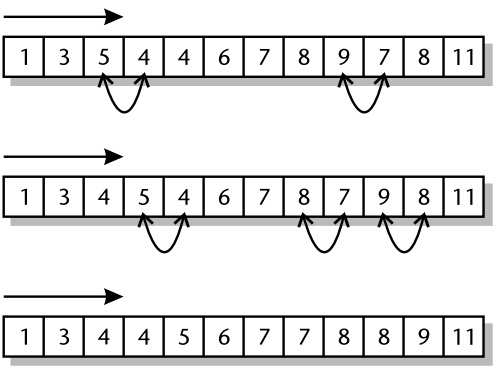


*Abbildung 7.1: Die Sortierroutine in schematischer Darstellung*

Der Rumpf unserer Klasse sieht wie folgt aus:

```
/* Beispiel Bubblesort: Rumpf */
/* Autor:   Frank Eller       */
/* Sprache: C#               */

public class Sorter
{

  public void Swap(ref int a, ref int b)
  {
    //Hier die Anweisungen zum Vertauschen
  }

  public void Sort(ref int[] theArray)
  {
    //Hier die Anweisungen zum Sortieren
  }
}
```

Kommen wir zunächst zu unserer `Swap()`-Methode, in der wir lediglich die zwei als Referenzparameter übergebenen Zahlen vertauschen:

```
/* Beispiel Bubblesort: Methode Swap */
/* Autor:   Frank Eller              */
/* Sprache: C#                       */

public void Swap(ref int a, ref int b)
{
  int c = a;
  a = b;
  b = c;
}
```

Diese Methode ist Ihnen sicherlich noch aus Kapitel 2 ein wenig in Erinnerung. Die eigentliche Funktionalität stellen wir nun in der Methode `Sort()` zur Verfügung. Dieser Methode wird, wie bereits an ihrer Deklaration zu ersehen, das gesamte Array, welches wir sortieren wollen, übergeben. Die Länge des Array können wir über die Eigenschaft `Length` erfahren. Wir wissen aber, dass die Indizes der Arrays in C# stets mit 0 beginnen, wenn wir also ein Array mit einer Länge von 10 Elementen haben, dürfen wir nur bis 9 zählen (wir beginnen dafür mit der Zählung bei 0). Andernfalls beschwert sich der Compiler. Die gesamte Routine zum Sortieren sieht damit folgendermaßen aus:

```
/* Beispiel Bubblesort: Methode Sort */
/* Autor:   Frank Eller              */
/* Sprache: C#                       */

public void Sort(ref int[] theArray)
{
  bool hasChanged = false;
  do
  {
    hasChanged = false;

    for (int i=1;i<theArray.Length;i++)
    {
      if (theArray[i-1]>theArray[i])
      {
        Swap(ref theArray[i-1],ref theArray[i]);
        hasChanged = true;
      }
    }
  } while (hasChanged);
}
```

Die Variable `hasChanged` kontrolliert, ob ein Austauschen innerhalb des Array stattgefunden hat. Die **do-while**-Schleife läuft, so lange dies der Fall ist. Der Rest ist bereits bekannt, eine simple **for**-Schleife dient dazu, das Array zu durchlaufen und den Austausch durchzuführen.

Nun fehlt nur noch eine `Main()`-Methode, damit wir das Programm auch starten können.

```
/* Beispiel Bubblesort: Methode Main() */
/* Autor:   Frank Eller               */
/* Sprache: C#                        */

public static void Main()
{
  int[] myArray = new int[10];
  Sorter S = new Sorter();

  for (int i=0;i<10;i++)
    myArray[i] = Console.ReadLine().ToInt32();

  S.Sort(ref myArray);

  for (int i=0;i<10;i++)
    Console.Write("{0},",myArray[i]);
}
```

Das wäre alles, was wir zum Sortieren benötigen. Das gesamte Programm im Zusammenhang nochmals hier:

```
/* Beispiel Bubblesort komplett */
/* Autor:   Frank Eller         */
/* Sprache: C#                  */


using System;

public class Sorter
{
  public void Swap(ref int a, ref int b)
  {
    int c = a;
    a = b;
    b = c;
  }

  public void Sort(ref int[] theArray)
```

```
  {
    bool hasChanged = false;
    do
    {
      hasChanged = false;

      for (int i=1;i<theArray.Length;i++)
      {
        if (theArray[i-1]>theArray[i])
        {
          Swap(ref theArray[i-1],ref theArray[i]);
          hasChanged = true;
        }
      }
    } while (hasChanged);
  }
}

public class Project1
{
  public static void Main()
  {
    int[] myArray = new int[10];
    Sorter S = new Sorter();

    Console.WriteLine("Geben Sie bitte 10 Zahlen ein:");

    for (int i=0;i<10;i++)
    {
      Console.Write("Zahl {0}: ",i+1);
      myArray[i] = Console.ReadLine().ToInt32();
    }
    S.Sort(ref myArray);

    for (int i=0;i<10;i++)
      Console.Write("{0},",myArray[i]);
  }
}
```

Das komplette Programm finden Sie ebenfalls auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_7\BUBBLESORT.

### 7.1.2 Mehrdimensionale Arrays

Ein Array kann auch mehrere Dimensionen besitzen. Das ist insbesondere dann nützlich, wenn Sie die Daten einer Tabelle in einem

Array speichern wollen. Sie können sehr einfach ein Array mit zwei Dimensionen erstellen:

```
int[,] theArray = new int[10,5];
```

Die Tabelle, die diesem Array entspricht, hätte dann zehn Zeilen und fünf Spalten (oder wahlweise auch zehn Spalten und fünf Zeilen, im Endeffekt ist das nicht relevant), es können also 50 Werte darin gespeichert werden. Aber auch hier gilt, dass der unterste Index des Array bei 0 liegt, d.h. der erste Wert unserer Tabelle findet sich in `theArray[0,0]` wieder, der letzte in `theArray[9,4]`.

***mehrere Dimensionen***

Natürlich ist dies nicht auf zwei Dimensionen beschränkt, auch drei, vier oder fünf Dimensionen sind denk- und machbar. Man sollte aber, falls man eine solche Konstruktion wirklich einmal benötigt, zunächst nach alternativen Lösungsmöglichkeiten suchen. Mehr als drei Dimensionen sind in der Regel unter Berücksichtigung der späteren Wartung des Programms nicht sinnvoll.

Mehrdimensionale Arrays sind immer gleichförmig, im obigen Beispiel hat also jede Zeile die gleiche Anzahl Spalten. Auch handelt es sich um ein einziges Array. Wie wir gleich sehen werden, gibt es auch noch eine andere Möglichkeit, ein mehrdimensionales Array zu erstellen.

### 7.1.3 Ungleichförmige Arrays

Ungleichförmige Arrays, im Original „Jagged“ Arrays, sind ebenfalls mehrdimensional, allerdings hat bei diesen Arrays nicht jede Zeile zwangsläufig die gleiche Anzahl Spalten. D.h. die Zeilen oder Spalten können zusammengefasst sein, so wie Sie es auch von Textverarbeitungsprogrammen her kennen.

Um das Ganze ein wenig verständlicher zu machen, stellen Sie sich Folgendes vor: Sie haben ein Array mit fünf Elementen deklariert. Diese Elemente sind aber ihrerseits ebenfalls Arrays, allerdings mit unterschiedlichen Größen. Damit ergibt sich, wenn man das erste Array als das Array für die Zeilen einer Tabelle betrachtet, eine Tabelle mit einer bestimmten Anzahl Zeilen, die aber ihrerseits eine unterschiedliche Anzahl Spalten besitzen. Die Tabelle ist ungleichförmig. Ein Beispiel soll Ihnen zeigen, wie dies funktioniert.

```
/* Beispiel Jagged Array */
/* Autor:   Frank Eller  */
/* Sprache: C#           */
```

```
public class ArrayTest
{
  public static void Main()
  {
    int[][] jArray = new int[5][];

    jArray[0] = new int[8];
    jArray[1] = new int[4];
    jArray[2] = new int[2];
    jArray[3] = new int[4];
  }
}
```

Die obigen Anweisungen haben eine ungleichförmige Tabelle konstruiert, in die Sie nun Daten eingeben könnten. Wie die Tabelle aufgezeichnet aussehen würde, zeigt Abbildung 7.1.

<table>
<tr><td>0,0</td><td>0,1</td><td>0,3</td><td>0,2</td><td>0,4</td><td>0,5</td><td>0,6</td><td>0,7</td></tr>
<tr><td colspan="2">1,0</td><td colspan="2">1,1</td><td colspan="2">1,2</td><td colspan="2">1,3</td></tr>
<tr><td colspan="4">2,0</td><td colspan="4">2,1</td></tr>
<tr><td colspan="2">3,0</td><td colspan="2">3,1</td><td colspan="2">3,2</td><td colspan="2">3,3</td></tr>
</table>

*Abbildung 7.2: Die erzeugte Tabelle als Grafik*

Ungleichförmige Arrays können Sie sich als Arrays in einem Array vorstellen. Dadurch wird die Funktionsweise dieses Features ein wenig einleuchtender. Während bei herkömmlichen mehrdimensionalen Arrays jedes Feld aus einem Wert des angegebenen Datentyps besteht, kann man sich ein ungleichförmiges Array wie ein eindimensionales Array vorstellen, bei dem jedes Feld wiederum aus einem Array besteht. Damit wird die Ungleichförmigkeit ermöglicht.

### 7.1.4 Arrays initialisieren

Ebenso wie andere Variablen können auch Arrays direkt bei der Deklaration initialisiert werden. Aufgrund der Menge der Daten, die dazu möglicherweise erforderlich sind, ist dies allerdings nur bei Arrays mit kleineren Dimensionen sinnvoll. Falls es sich um eine große Anzahl Werte handelt, sollten Sie sich überlegen, ob es nicht sinnvoller bzw. zeitsparender wäre, für die Initialisierung eine kleine Methode zu schreiben.

***eindimensionale Arrays initialisieren***

Wenn Sie ein Array direkt mit Werten bestücken, müssen Sie **new** nicht benutzen. Stattdessen schreiben Sie die Werte in geschweiften Klammern als Zuweisung hinter die Deklaration:

```
int[] theArray = {15,17,19,21,23};
```

Diese Anweisung bewirkt dasselbe wie die Anweisungen

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

theArray[0] = 15;
theArray[1] = 17;
theArray[2] = 19;
theArray[3] = 21;
theArray[4] = 23;
```

Sie sehen, bei kleineren Arrays spart die direkte Deklaration eine Menge Schreibarbeit ein.

***mehrdimensionale Arrays initialisieren***

Bei mehrdimensionalen Arrays sieht das Ganze ein wenig anders aus. Zwar können Sie auch diese gleich bei der Deklaration mit Werten beladen, allerdings muss der Compiler genau unterscheiden können, welcher Wert wohin soll. Es genügt also nicht, einfach alle Werte hintereinander zu schreiben. Stattdessen schreiben Sie zusammengehörige Werte in geschweifte Klammern und diesen ganzen Ausdruck nochmals in geschweifte Klammern. Auch hier zum besseren Verständnis eine Beispielanweisung.

```
int[,] theArray = {{1,1},{2,2},{3,3}};
```

Diese einzelne Anweisung kann durch folgende Anweisungen ersetzt werden:

```
int[,] theArray = new int[3,2];

theArray[0,0] = 1;
theArray[0,1] = 1;
theArray[1,0] = 2;
theArray[1,1] = 2;
theArray[2,0] = 3;
theArray[2,1] = 3;
```

Auch hier wieder eine Einsparung an Schreibarbeit. Wie bereits angesprochen, die direkte Initialisierung von Arrays kann durchaus Vorteile bringen.

### 7.1.5 Die foreach-Schleife

Im Zusammenhang mit Arrays kommen wir zu einer besonderen Form einer Schleife, die im vorhergehenden Kapitel noch nicht angesprochen worden war, nämlich der **foreach**-Schleife. Diese Schleifenkonstruktion dient dazu, die Elemente eines Array oder eines anderen Listentyps durchzugehen und damit zu arbeiten. Sie erreichen mit dieser Schleifenform alle enthaltenen Elemente einer Liste, z.B. um eine Kontrolle durchzuführen. Die Syntax der **foreach**-Schleife lautet wie folgt:

*Syntax*

```
foreach (Datentyp Bezeichner in (Liste))
{
  //Anweisungen
}
```

Der Datentyp, der in der Schleife verwendet wird, muss natürlich dem Datentyp entsprechen, der auch für das Array benutzt wurde. Was die Schleife nämlich tut, ist, jedes Element des Arrays in einer Variable mit dem im Schleifenkopf angegebenen Namen und Datentyp abzulegen, so dass Sie damit arbeiten können. Ist der Schleifenblock abgearbeitet, wird weitergeschaltet, das nächste Element aus dem Array entnommen und die Anweisungen werden erneut durchgeführt.

```
/* Beispiel foreach-Schleife 1 */
/* Autor:   Frank Eller       */
/* Sprache: C#               */

using System;

public class Text
{
  public static void Main()
  {
    int[] theArray = new int[10];

    for (int i=0;i<10;i++)
      theArray[i] = i;

    foreach (int u in theArray)
    {
      if (u<5)
        Console.WriteLine(u);
    }
  }
}
```

In diesem Beispiel würden die Zahlen von 0 bis 4 auf dem Bildschirm ausgegeben. Ab dann würde die Bedingung der **if**-Schleife nicht mehr erfüllt und die nachfolgende Ausgabeanweisung nicht mehr ausgeführt. Sie finden das Programm auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_7\FOREACH1.

Bei großen Listen bzw. Arrays kann die Ausführung einer solchen Schleife schon mal einige Zeit dauern. Aber man kann das auch abkürzen, z.B. beim Suchen durch ein Array – es gibt ja noch die **break**-Anweisung.

```
/* Beispiel foreach-Schleife 2 */
/* Autor:   Frank Eller       */
/* Sprache: C#               */

using System;

public class Text
{
  public static void Main()
  {
    int[] theArray = new int[10];

    for (int i=0;i<10;i++)
      theArray[i] = i;

    foreach (int u in theArray)
    {
      if (u=5)
        break;
      else
        Console.WriteLine(u);
    }
  }
}
```

Wie bei den anderen Schleifentypen erzwingt die Anweisung **break** das Verlassen des Schleifenblocks. Und da es sich bei **foreach** ebenfalls um eine Schleife handelt, können wir **break** natürlich auch benutzen. Das Programm verhält sich ebenso wie das vorhergehende Beispiel. Sie finden es natürlich auch auf der CD im Vezeichnis BEISPIELE\KAPITEL_7\FOREACH2.

## 7.2 Structs

Die so genannten **structs** sind Strukturen in C#, die sowohl Daten als auch Methoden enthalten können. Sie werden genauso deklariert wie Klassen, können ebenfalls einen Konstruktor enthalten und auch die Syntax für die Deklaration ist die gleiche. Alles, was Sie tun müssen, um eine Klasse zu einem **struct** zu machen, ist, das reservierte Wort **class** mit dem reservierten Wort **struct** zu vertauschen.

***Wozu structs?***

Sie werden sich sicherlich fragen, warum überhaupt einen **struct** deklarieren, wenn es ohnehin das Gleiche ist wie eine Klasse? Nun, es ist nicht ganz das Gleiche. Klassen sind Referenztypen, wenn Sie ein neues Objekt (eine Instanz der Klasse) erstellen, dann wird Speicher dafür reserviert und eine Referenz auf diesen Speicher in der Variablen gespeichert, die das Objekt darstellt. Anders verhält es sich bei einem **struct**, der ein Wertetyp ist. Daher sind **structs** für kleine Objekte geeignet, die nicht viel Speicher benötigen, aber in großer Zahl innerhalb der Applikation verwendet werden.

Als Faustregel soll gelten: Ein **struct** ist dann effizienter als eine Klasse, wenn seine Größe weniger als 16 Byte beträgt. Ein Beispiel für einen **struct** wäre die Angabe der Koordinaten eines Punkts auf dem Bildschirm:

```
/* Beispiel structs 1   */
/* Autor:   Frank Eller */
/* Sprache: C#          */


public struct Point
{
  public int x;
  public int y;

  public Point();
  {
    x = 0;
    y = 0;
  }
}
```

Wie Sie sehen, hat auch ein **struct** einen Konstruktor – ebenso wie es bei Klassen der Fall ist. Allerdings ist ein **struct** ein Wertetyp, während eine Klasse ein Referenztyp ist.

Eingesetzt in einem Programm sieht unser **struct** folgendermaßen aus:

```
/* Beispiel structs 2   */
/* Autor:   Frank Eller */
/* Sprache: C#          */

public struct Point
{
  public int x;
  public int y;

  public Point();
  {
    x = 0;
    y = 0;
  }
}

public class StructTest
{
  public static void Main()
  {
    Point coords;
    coords.x = 100;
    coords.y = 150;

    Console.WriteLine("Der Punkt liegt bei {0}/{1}",
                      coords.x,coords.y);
  }
}
```

Achten Sie darauf, die Felder des **struct** nicht gleich bei der Deklaration zu initialisieren. Der Compiler quittiert diesen Versuch mit einer Fehlermeldung.

## 7.3 Aufzählungen

In C# gibt es die Möglichkeit, Aufzählungen, so genannte *enums*, zu verwenden. Dabei handelt es sich um einen Mengentyp, bei dem die enthaltenen Elemente den gleichen Wertetyp besitzen. Eine solche Aufzählung fungiert dann als eigener Datentyp, den Sie wie die anderen Datentypen auch in Ihrer Anwendung verwenden können.

***Syntax***

```
[Modifikator] enum Bezeichner [: Typ]
  {
    Wert 1, Wert 2 ... , Wert n
  };
```

### 7.3.1 Standard-Aufzählungen

Standardmäßig wird vom Compiler automatisch der Datentyp **int** für die enthaltenen Elemente benutzt. Das erste Element beginnt wie bei Programmiersprachen üblich mit 0, das zweite hat den Wert 1 usw. Bei der folgenden Deklaration einer Aufzählung für die Wochentage hätte also der Sonntag die 0, der Montag die 1 usw.

```
/* Beispiel Aufzählungstypen 1   */
/* Autor:   Frank Eller          */
/* Sprache: C#                   */

public enum WeekDays
  {
    Sonntag, Montag, Dienstag, Mittwoch,
    Donnerstag, Freitag, Samstag
  };
```

***Wertevorgaben***

Oftmals ist es nicht erwünscht oder einfach auch für die spätere Programmierung sinnvoller, wenn die Werte der Aufzählungselemente nicht bei 0 beginnen. Sie können den Compiler anweisen, einen anderen Wert als Startwert zu benutzen. In diesem Fall weisen Sie einfach dem ersten Element einen Wert zu, die nächsten Elemente erhalten dann den jeweils nächsten Wert:

```
/* Beispiel Aufzählungstypen 2   */
/* Autor:   Frank Eller          */
/* Sprache: C#                   */

public enum WeekDays
  {
    Sonntag = 1, Montag, Dienstag, Mittwoch,
    Donnerstag, Freitag, Samstag
  };
```

In diesem Fall erhält der Sonntag den Wert 1, da er direkt zugewiesen wurde. Der Montag als nächstes Element erhält automatisch den Wert 2, der Dienstag den Wert 3 usw.

Sie können aber auch noch weiter gehen und jedem Element einen Wert zuweisen. Im Falle der Wochentage macht dies zwar nicht viel Sinn, es könnte aber von Vorteil sein, wenn Sie eine eigene Aufzählung verwenden. Als Beispiel möchte ich hier die Monate des Jahres verwenden. Sie könnten eine Aufzählung erstellen und jedem Monat einen Wert zuweisen, der der Anzahl der Tage des Monats entspricht. Dass es dabei zu Überschneidungen kommt, macht dem Compiler nichts aus, er verarbeitet es klaglos – für ihn besteht eine Aufzählung

nur aus einer Menge von Variablen des gleichen Typs, die nun einmal zusammengefasst sind. Ihr Wert ist dem Compiler egal:

```
/* Beispiel Aufzählungstypen 3   */
/* Autor:   Frank Eller          */
/* Sprache: C#                   */

public enum Months
  {
    Januar = 31,Februar = 28, Maerz = 31, April = 30,
    Mai = 31, Juni = 30, Juli = 31, August = 31,
    September = 30, Oktober = 31, November = 30,
    Dezember = 31
  };
```

***Einen Datentyp festlegen***

Bei der Deklaration eines **enum** sind Sie nicht auf den Datentyp **int** beschränkt. Sie können durchaus einen anderen integralen Datentyp (außer **char**) verwenden. Das folgende Beispiel zeigt eine Aufzählung basierend auf dem Datentyp **byte**:

```
/* Beispiel Aufzählungstypen 4   */
/* Autor:   Frank Eller          */
/* Sprache: C#                   */

public enum WeekDays : byte
  {
    Sonntag = 1, Montag, Dienstag, Mittwoch,
    Donnerstag, Freitag, Samstag
  };
```

Es wurde sowohl der Datentyp **byte** festgelegt als auch der Beginn der verwendeten Werte auf 1.

Bei dieser Art von Aufzählungen ist eine Sache allerdings nicht möglich, nämlich die Verknüpfung von Werten, so dass man z.B. zwei oder drei im Aufzählungstyp angegebene Elemente zusammengefasst verwenden könnte. Stattdessen ist immer nur die Angabe eines der enthaltenen Elemente möglich.

Allerdings gibt es in C# eine spezielle Art von Aufzählungstyp, der auch die Verknüpfung mehrerer Elemente erlaubt. Da es sich bei einer solchen Funktionsweise um eine Art von Flags handelt, trägt der entsprechende Aufzählungstyp auch den Namen *Flag-Enum*.

### 7.3.2 Flag-Enums

Dass es sich bei unserer Aufzählung um ein Flag-Enum handeln soll, müssen wir dem Compiler über ein so genanntes Compiler-Attribut angeben. Vor der eigentlichen Deklaration des Aufzählungstyps schreiben wir in eckigen Klammern das zu verwendende Attribut, in diesem Fall das Attribut *Flags*.

```
/* Beispiel Flag-Enums 1 */
/* Autor:   Frank Eller  */
/* Sprache: C#           */

[Flags]
public enum PrinterSettings
{
  PrintPageNr   = 1,
  PrintHeader   = 2,
  PrintFooter   = 4,
  PrintTitle    = 8,
  PrintGraphics = 16
};
```

Die Werte für die einzelnen Elemente der Aufzählung müssen Bitwerte sein, damit der Compiler die einzelnen gesetzten Bits auch unterscheiden kann. Eine andere Möglichkeit, diese Werte anzugeben, wäre diese:

```
/* Beispiel Flag-Enums 2 */
/* Autor:   Frank Eller  */
/* Sprache: C#           */


[Flags]
public enum PrinterSettings
{
  PrintPageNr   = 0x00000001,
  PrintHeader   = 0x00000002,
  PrintFooter   = 0x00000004,
  PrintTitle    = 0x00000008,
  PrintGraphics = 0x00000010
};
```

In diesem Fall werden die Bitwerte als Hexadezimalwerte angegeben (Hexadezimal 10 entspricht dezimal 16).

Innerhalb einer `Main()`-Funktion könnten diese Werte jetzt miteinander verknüpft werden:

```
/* Beispiel Flag-Enums 3 */
/* Autor:   Frank Eller  */
/* Sprache: C#          */

using System;

[Flags]
public enum PrinterSettings
{
  PrintPageNr   = 0x00000001,
  PrintHeader   = 0x00000002,
  PrintFooter   = 0x00000004,
  PrintTitle    = 0x00000008,
  PrintGraphics = 0x00000010
};

public class MainProg
{
  public static PrinterSettings SetFlags(int theType)
  {
    select theType
    {
      case 1: return PrintPageNr;
              break;
      case 2: return PrintHeader | PrintFooter;
              break;
      ...
    }
  }

  public static void DoPrint(PrinterSettings prt)
  {
    //Funktionen zum Ausdrucken entsprechend
    //der gesetzten Flags
  }

  public static void Main();
  {
    a = Console.ReadLine();
    DoPrint(SetFlags(a));
  }
```

Sie erinnern sich: Da `Main()` eine statische Methode ist, können nur andere statische Methoden aufgerufen werden, wir müssen also alle verwendeten Methoden statisch machen. Für das Beispiel hier ist das allerdings kein Problem. Wenn Sie es umgehen wollen, können Sie für die diversen Methoden auch eine eigene Klasse deklarieren.

## 7.4 Kontrollfragen

Wiederum möchte ich einige Fragen dazu benutzen, Ihnen bei der Vertiefung des Gelernten zu helfen.

1. Bis zu welcher Größe ist ein `struct` effektiv?
2. Was ist der Unterschied zwischen einem `struct` und einer Klasse?
3. Mit welchem Wert beginnt standardmäßig eine Aufzählung?
4. Welche Arten von Aufzählungstypen gibt es?
5. Wie kann der Datentyp, der für eine Aufzählung verwendet wird, angegeben werden?
6. Auf welche Art können den Elementen einer Aufzählung unterschiedliche Werte zugewiesen werden?

## 7.5 Übungen

### Übung 1

Erstellen Sie eine neue Klasse. Programmieren Sie eine Methode, mit deren Hilfe der größte Wert eines Array aus Integer-Werten ermittelt werden kann. Die Methode soll die Position des Wertes im Array zurückliefern.

### Übung 2

Fügen Sie der Klasse eine Methode hinzu, die die Position des kleinsten Wertes in einem Integer-Array zurückliefert.

### Übung 3

Fügen Sie der Klasse eine Methode hinzu, die die Summe aller Werte des Integer-Array zurückliefert.

### Übung 4

Fügen Sie der Klasse eine Methode hinzu, die den Durchschnitt aller Werte im Array zurückliefert.

# 8 Vererbung

Vererbung und Polymorphie sind zwei Grundprinzipien der objektorientierten Programmierung. Diese Art der Programmierung ist eigentlich recht eng an die Vorgaben der Natur angelehnt. So ist es möglich, dass eine Klasse Nachfolger hat, von jedem Nachfolger aber auch auf den Vorgänger geschlossen werden kann. Auch in C#, ebenso wie in C++, Java oder Delphi, ist dieses Grundprinzip der objektorientierten Programmierung enthalten, ebenso konsequent wie alles andere.

## 8.1 Vererbung von Klassen

***Vererbung***

Bei der Vererbung wird eine neue Klasse von einer bestehenden Klasse abgeleitet und kann dann erweitert werden. Dabei „erbt“ sie alle Eigenschaften und Methoden der Basisklasse, die, je nachdem, wie sie deklariert wurden, überschrieben oder erweitert werden können. Ebenso wie in der Natur kann in C# jede Klasse zwar beliebig viele Nachfolger besitzen, aber sie kann lediglich einen Vorgänger, eine Basisklasse haben.

***Polymorphie***

Polymorphie ist eine Eigenschaft, die sich auf den umgekehrten Weg bezieht. Wird eine Klasse von einer anderen Klasse abgeleitet, so kann eine Instanz der neuen Klasse auch der Elternklasse zugewiesen werden. Wir haben das schon öfter getan, ohne es zu bemerken. Wenn wir die Methode `WriteLine()` benutzt haben, um etwas auszugeben, konnten wir Platzhalter benutzen und die auszugebenden Werte zusätzlich angeben. Dabei ist es möglich, jeden beliebigen Datentyp zu verwenden. Der Grund hierfür ist, dass der Datentyp der übergebenen Parameter `object` ist, der Basistyp aller Klassen in C#. Es kann also jeder andere Datentyp aufgenommen werden, da alle von `object` abgeleitet sind.

Das Ableiten einer Klasse geschieht über den :-Operator. Die neue Klasse wird mittels eines Doppelpunkts von der Klasse getrennt, von der sie abgeleitet wird. Die Syntax unserer bisherigen Klassendeklarationen wird also für diesen Fall erweitert:

***Syntax***

```
[Modifikator] class Klasse : Elternklasse
{
  //Attribute ...
}
```

Nach dem Ableiten besitzt die neue Klasse alle Eigenschaften und Methoden der Elternklasse, auch wenn diese nicht explizit angegeben sind. Die vorhandenen Methoden können jedoch auch angepasst werden. Sie können sowohl bestehende Methoden verbergen als auch überschreiben oder eine neue Methode mit dem gleichen Namen einer bestehenden Methode deklarieren, aber eine andere Funktionalität darin unterbringen. Es kann in diesem Fall auf beide Methoden zugegriffen werden.

### 8.1.1 Verbergen von Methoden

Das Verbergen einer Methode bietet sich dann an, wenn Sie von einer Klasse ableiten, die Sie nicht selbst geschrieben haben. Innerhalb einer Klasse müssen Methoden, die überschrieben werden können, entsprechend gekennzeichnet sein (mit dem Modifikator **virtual**), wovon Sie aber nicht ausgehen können, da Sie nicht wissen, was der ursprüngliche Programmierer einer Klasse denn so angestellt hat. Falls also eine von Ihnen implementierte Methode den gleichen Namen besitzt wie eine Methode der Basisklasse, können Sie den Compiler anweisen, die ursprüngliche Methode durch die neue zu verbergen. Das folgende Beispiel zeigt eine solche Situation, wobei die zu verbergende Methode `GetAdr()` heißt.

```
/* Beispielklassen Verbergen */
/* Autor:   Frank Eller      */
/* Sprache: C#               */

using System;

public class cAdresse
{
  protected string name;
  protected string strasse;
  protected string plz;
  protected string ort;
```

```
  public cAdresse()
  {
    this.name    = "";
    this.strasse = "";
    this.plz     = "";
    this.ort     = "";
  }

  public void SetAdr(string n,string s,string p,string o)
  {
    this.name = n;
    this.strasse = s;
    this.plz = p;
    this.ort = o;
  }

  public string GetAdr()
  {
    return string.Format("{0}\n{1}\n\n{2} {3}",
                         name,strasse,plz,ort);
  }
}


public class neuAdresse : cAdresse
{
  public neuAdresse()
  {
    //Konstruktoren werden nicht vererbt
  }

  new public string GetAdr()
  {
    return string.Format("{0}, {1} {2}",name,plz,ort);
  }
}
```

Sie finden das Programm auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_8\VERBERGEN1, natürlich mit einer entsprechenden Hauptklasse, so dass Sie die Funktion auch testen können.

Die Klasse `neuAdresse` ist die abgeleitete Klasse, die Klasse `cAdresse` die Basisklasse. Die Felder der Basisklasse werden hier übernommen und müssen nicht mehr deklariert werden. Wir möchten aber, dass Instanzen der abgeleiteten Klasse eine andere Ausgabe zur Verfügung stellen als in der ursprünglichen Klasse vorgesehen. Dazu verbergen wir die bereits bestehende Methode `GetAdr()` und schreiben sie neu.

*new*

Wenn eine bestehende Methode verborgen werden soll, so dass in einer Instanz der neu erstellten Klasse auch die neue Methode und nicht die ursprüngliche aufgerufen wird, müssen wir das Schlüsselwort **new** verwenden. Die Bedeutung ist aber nicht die gleiche wie beim Erzeugen eines Objekts, wo ja das gleiche Schlüsselwort verwendet wird. Während **new** dort die Bedeutung „Erstelle eine neue Kopie von ...“ hat, ist die Bedeutung jetzt „Stelle eine neue Methode mit gleichem Namen zur Verfügung“. Wenn diese Methode nun aufgerufen wird, wird nur die neue Methode benutzt und nicht mehr die ursprüngliche.

Selbstverständlich können Sie immer noch eine Instanz der Basisklasse erzeugen und auf deren Methode `GetAdr()` zugreifen. Lediglich wenn Sie eine Instanz der abgeleiteten Klasse benutzen, wird auch die neue Methode benutzt. Das macht durchaus Sinn, denn ansonsten könnte man die Basisklasse ja nicht mehr benutzen.

Der Modifikator **new** zeigt an, dass eine Methode neu deklariert wurde. Die Basismethode wird damit überschrieben. Die neue Methode wird allerdings nur dann aufgerufen, wenn es sich beim Aufruf um eine Instanz der neuen Klasse handelt.

*protected*

Der Modifikator **protected** in der Deklaration der Felder unserer Basisklasse bedeutet, dass auch diese Felder nur innerhalb der Klasse selbst oder von abgeleiteten Klassen aus zugegriffen werden kann. Fremde Klassen allerdings können nicht darauf zugreifen. Für unsere abgeleitete Klasse aber ist sichergestellt, dass auch sie die deklarierten Variablen benutzen kann.

### 8.1.2 Überschreiben von Methoden

Methoden, die überschrieben werden können, müssen in der Basisklasse entsprechend gekennzeichnet sein, und auch in der abgeleiteten Klasse müssen Sie dem Compiler angeben, dass die entsprechende Methode überschrieben werden soll. Die beiden dafür zuständigen Modifikatoren sind **virtual** und **override**.

*virtual und override*

**virtual** wird in der Klasse benutzt, von der abgeleitet werden soll. Methoden, die überschrieben werden können, werden als **virtual** deklariert. Wenn Sie solche Methoden überschreiben, müssen Sie in der abgeleiteten Klasse auch den Modifikator **override** benutzen.

*base*

Der Vorteil hierbei ist, dass immer noch auf die Methode der Basisklasse zugegriffen werden kann. Hierzu verwenden Sie das reservierte Wort **base**, das so ähnlich wirkt wie das uns schon bekannte **this**. Während **this** auf die Felder bzw. Methoden der aktuellen Instanz ei-

ner Klasse zugreift, können Sie mit **base** auf Methoden der Elternklasse zugreifen. Um dies deutlich zu machen, bauen wir unser Beispiel ein wenig um:

```
/* Beispielklassen Überschreiben 1 */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

using System;

public class cAdresse
{
  protected string name;
  protected string strasse;
  protected string plz;
  protected string ort;
  protected string tel;  //Neue Variable

  public cAdresse()
  {
    this.name    = "";
    this.strasse = "";
    this.plz     = "";
    this.ort     = "";
    this.tel     = "";
  }

  public void SetAdr(string n,string s,
                     string p,string o, string t)
  {
    this.name    = n;
    this.strasse = s;
    this.plz     = p;
    this.ort     = o;
    this.tel     = t;
  }

  public virtual string GetAdr()
  {
    return string.Format("{0}\n{1}\n\n{2} {3}",
                         name,strasse,plz,ort);
  }
}
```

```
public class neuAdresse : cAdresse
{

  public neuAdresse()
  {
    //Konstruktoren werden nicht vererbt
  }

  public override string GetAdr()
  {
    string x = base.GetAdr();
    return x+string.Format("\n{0}",tel);
  }
}
```

Das komplette Beispielprogramm finden Sie auf der CD im Verzeichnis BEISPIELE\KAPITEL_8\ÜBERSCHREIBEN1.

In der Methode `GetAdr()` wird nun zunächst die ursprüngliche Methode aufgerufen, bevor dem Ergebniswert ein zusätzliches Element hinzugefügt wird. Abbildung 8.1 zeigt schematisch, wie die **base**-Anweisung funktioniert.

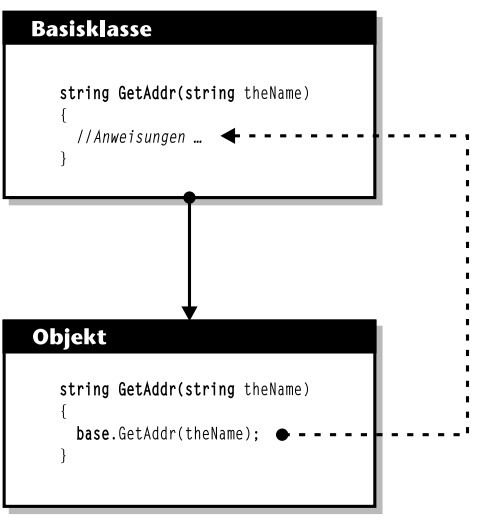


*Abbildung 8.1: Aufruf einer Methode der Basisklasse*

Wenn Sie Ihre selbst geschriebenen Klassen anderen Programmierern zur Verfügung stellen wollen, dürfen Sie natürlich nicht vergessen, diejenigen Methoden als **virtual** zu deklarieren, die von einer etwaigen abgeleiteten Klasse neu definiert werden könnten.

Wenn andere Programmierer nun eine neue Methode deklarieren, die den gleichen Namen und die gleichen Parameter wie eine Methode der Ursprungsklasse besitzt und diese Methode als virtual deklariert ist, erhält der Programmierer vom Compiler die Mitteilung, dass er **override** benutzen muss. Der Compiler hilft also immer dabei, keine Fehler zu machen.

Der Modifikator **virtual** zeigt an, dass die so bezeichnete Methode überschrieben werden kann. Mit dem Modifikator **override** wird angezeigt, dass die so bezeichnete Methode eine neue Deklaration einer bereits bestehenden Methode ist, diese aber nicht verbirgt.

***Standardmethoden überschreiben***

Damit wäre eigentlich schon alles zum Überladen und Überschreiben von Methoden gesagt. Wichtig wäre noch, dass auch viele Methoden, die bereits standardmäßig zur Verfügung gestellt werden, ebenfalls als **virtual** deklariert sind. Da alle neuen Klassen von der Basisklasse `Object` abgeleitet sind, können wir auch bereits vorhandene Methode überschreiben und somit auch solche Standardmethoden verwenden. Ein Beispiel wäre z.B. die Methode `ToString`, die wir schon häufiger verwendet haben.

```
/* Beispielklasse Überschreiben 2 */
/* Autor:   Frank Eller          */
/* Sprache: C#                   */

using System;

public class cAdresse
{
  protected string name;
  protected string strasse;
  protected string plz;
  protected string ort;
  protected string tel;

  public cAdresse()
  {
    this.name    = "";
    this.strasse = "";
    this.plz     = "";
    this.ort     = "";
    this.tel     = "";
  }
```

```
  public void SetAdr(string n,string s,string p,string o,
string t)
  {
    this.name    = n;
    this.strasse = s;
    this.plz     = p;
    this.ort     = o;
    this.tel     = t;
  }


  public override string ToString()
  {
    return string.Format("{0}\n{1}\n\n{2} {3}",name,strasse,plz,ort);
  }
}
```

In diesem Fall überschreiben wir die Methode `ToString()`, die ohnehin zur Verfügung steht, und ermöglichen somit eine Rückgabe unserer eigenen Werte in dem von uns gewünschten Format mittels einer bekannten Standardmethode. `ToString()` steht natürlich zur Verfügung, weil die Methode bereits von der Basisklasse **object** zur Verfügung gestellt wird, von der alle anderen Klassen abstammen.

Achten Sie beim Ableiten von Klassen immer darauf, dass die Methoden, die Sie in der abgeleiteten Klasse überschrieben haben, die gleiche oder eine geringere Sichtbarkeit als die entsprechenden Methoden der Basisklasse besitzen. Wenn die Sichtbarkeit größer ist, meldet der Compiler einen Fehler.

### 8.1.3 Den Basis-Konstruktor aufrufen

Wie wir bereits gesehen haben, ist es nicht möglich, den Konstruktor einer Klasse zu vererben, d.h. wenn wir eine eigene Klasse erzeugen, müssen wir auch einen Konstruktor für die neue Klasse zur Verfügung stellen. Oftmals ist es aber sinnvoll, im neuen Konstruktor den ursprünglichen Konstruktor ebenfalls aufzurufen. Dieser Vorgang kann automatisiert werden, ebenfalls durch Verwendung des reservierten Worts **base**.

```
/* Beispielklassen Konstruktor 1 */
/* Autor:   Frank Eller          */
/* Sprache: C#                   */

using System;
```

```
public class cAdresse
{
  protected string name;
  protected string strasse;
  protected string plz;
  protected string ort;

  public cAdresse()
  {
    this.name    = "";
    this.strasse = "";
    this.plz     = "";
    this.ort     = "";
  }

  public void SetAdr(string n,string s,string p,string o)
  {
    this.name = n;
    this.strasse = s;
    this.plz = p;
    this.ort = o;
  }

  public string GetAdr()
  {
    return string.Format("{0}\n{1}\n\n{2} {3}",
                         name,strasse,plz,ort);
  }
}


public class neuAdresse : cAdresse
{
  public neuAdresse() : base()
  {
    //Basis-Konstruktor wird automatisch aufgerufen
  }

  new public string GetAdr()
  {
    return string.Format("{0}, {1} {2}",name,plz,ort);
  }
}
```

Das Beispielprogramm finden Sie auf der CD im Verzeichnis BEISPIELE\KAPITEL_8\KONSTRUKTOR1.

Durch die Konstruktion `public neuAdresse : base()` wird automatisch der Konstruktor der ursprünglichen Klasse aufgerufen, wenn ein neues Objekt unserer Klasse erzeugt wird. Auch bei mehreren Konstruktoren, die sich ja durch die Anzahl der Parameter unterscheiden, funktioniert dies. In diesem Fall wird ebenfalls das reservierte Wort **base** benutzt, die Parameter werden mit übergeben. Ein Beispiel sehen Sie hier:

```
/* Beispielklassen Konstruktor 2 */
/* Autor:   Frank Eller          */
/* Sprache: C#                   */

using System;

public class cAdresse
{
  protected string name;
  protected string strasse;
  protected string plz;
  protected string ort;

  public cAdresse()
  {
    this.name    = "";
    this.strasse = "";
    this.plz     = "";
    this.ort     = "";
  }

  public cAdresse(string a,string b,string c,string d)
  {
    this.name    = a;
    this.strasse = b;
    this.plz     = c;
    this.ort     = d;
  }

  public void SetAdr(string n,string s,string p,string o)
  {
    this.name    = n;
    this.strasse = s;
    this.plz     = p;
    this.ort     = o;
  }
```

```
  public string GetAdr()
  {
    return string.Format("{0}\n{1}\n\n{2} {3}",
                         name,strasse,plz,ort);
  }
}


public class neuAdresse : cAdresse
{
  public neuAdresse() : base()
  {
    //Basis-Konstruktor wird automatisch aufgerufen
  }

  public neuAdresse(string a,string b,
                    string c,string d): base(a,b,c,d)
  {
    //Basis-Konstruktor wird automatisch aufgerufen
  }

  new public string GetAdr()
  {
    return string.Format("{0}, {1} {2}",name,plz,ort);
  }
}
```

Das reservierte Wort **base** darf nur innerhalb eines Konstruktors oder innerhalb von Instanzmethoden verwendet werden, nicht bei statischen Methoden. Der Grund hierfür ist klar, denn eine statische Methode ist Bestandteil der Klasse selbst und kennt somit keinen Vorgänger. Ein Aufruf einer Methode der Ursprungsklasse ist somit nicht möglich.

Auch das obige Beispiel finden Sie auf der CD im Verzeichnis BEISPIELE\KAPITEL_8\KONSTRUKTOR2.

### 8.1.4 Abstrakte Klassen

C# bietet die Möglichkeit, so genannte *abstrakte Klassen* zu deklarieren. Abstrakte Klassen beinhalten zwar ebenso wie eine herkömmliche Klasse auch Methoden und Felder, allerdings ist es nicht zwingend notwendig, dass diese auch eine Funktionalität enthalten. Stattdessen werden sie als **abstract** deklariert, worauf die abgeleitete Klasse dazu gezwungen ist, eben diese als **abstract** deklarierten Me-

thoden zu implementieren und eigenen Code dafür zur Verfügung zu stellen.

Aufgrund der fehlenden Funktionalität können von abstrakten Klassen wie gesagt keine Instanzen erstellt werden, es muss zunächst eine andere Klasse davon abgeleitet werden. In dieser müssen alle als **abstract** deklarierten Methoden implementiert werden, wiederum mit dem Modifikator **override**.

```
/* Beispielklasse abstrakte Klassen */
/* Autor:   Frank Eller             */
/* Sprache: C#                      */

using System;

public abstract class Arbeitnehmer
{
  protected string name;
  protected string vorname;

  public abstract string GetBeruf();

  public string GetName()
  {
    return string.Format("{0}, {1}",name,vorname);
  }

  public void SetName(string n, string v)
  {
    name    = n;
    vorname = v;
  }
}

public class Elektriker : Arbeitnehmer
{
  public override string GetBeruf()
  {
    return ("Elektriker");
  }
}
```

Das Beispielprogramm finden Sie auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_8\ABSTRAKT.

Die Klasse `Elektriker` wurde von der Basisklasse `Arbeitnehmer` abgeleitet, beinhaltet nun also auch die in der Basisklasse deklarierten Felder `name`

und vorname. Allerdings muss die Methode GetBeruf(), die als **abstract** deklariert ist, in der abgeleiteten Klasse implementiert, also mit Funktionalität versehen werden.

Als **abstract** gekennzeichnete Methoden müssen in der Klasse, die von einer abstrakten Klasse abgeleitet ist, implementiert werden. Das gilt auch dann, wenn es sich um Methoden handelt, die in der neuen Klasse nicht benötigt werden.

### 8.1.5 Versiegelte Klassen

Es gibt in C# auch Klassen, von denen keine weitere Klasse abgeleitet werden darf. Das wäre dann das Gegenteil zur abstrakten Klasse, bei der es ja unumgänglich ist, dass abgeleitet wird.

Klassen, von denen man nicht ableiten kann, heißen auch *versiegelte Klassen*, ebenso wie der Modifikator, der dafür verwendet wird, **sealed** (engl. für versiegelt) heißt. Die bekannteste versiegelte Klasse in C# ist **string**. Schade ist nur, dass man dem Programmierer auf diese Weise die Möglichkeit genommen hat, eine eigene **string**-Klasse abzuleiten, die möglicherweise eine andere Funktionalität bei der Verwendung von Operatoren zur Verfügung stellen könnte.

Wenn wir als Beispiel unsere Klasse cAdress versiegeln wollen, würde das also folgendermaßen aussehen:

```
/* Beispielklasse Versiegeln */
/* Autor:   Frank Eller      */
/* Sprache: C#              */

using System;

public sealed class cAdresse
{
  private string name;
  private string strasse;
  private string plz;
  private string ort;

  public cAdresse()
  {
    name    = "";
    strasse = "";
    plz     = "";
    ort     = "";
  }
```

```
  public void SetAdr(string n,string s,string p,string o)
  {
    name    = n;
    strasse = s;
    plz     = p;
    ort     = o;
  }

  public string ToString()
  {
    return string.Format("{0}\n{1}\n\n{2} {3}",
                         name,strasse,plz,ort);
  }
}
```

Wie Sie sehen, finden Sie in einer versiegelten Klasse auch nicht mehr solche Modifikatoren wie **virtual**, **protected**, **override** oder ähnliche, die mit Vererbung zu tun haben. Es würde ja keinen Sinn machen, eine Methode als **virtual** zu deklarieren, wenn von der Klasse ohnehin nicht abgeleitet werden kann. Und ein Modifikator wie **protected** hätte die gleiche Wirkung wie **private**, und zwar aus dem gleichen Grund.

Da versiegelte Klassen nicht abgeleitet werden können, darf eine solche Klasse natürlich keinerlei abstrakte Methoden beinhalten. Das würde nämlich zur Ableitung zwingen, dies ist aber nicht möglich. Ebenso unmöglich sind alle Modifikatoren, die mit Vererbung zu tun haben, z.B. **virtual**. Da nicht abgeleitet werden kann, sind viertuelle Methoden unsinnig.

## 8.2 Interfaces

Interfaces funktionieren ähnlich wie abstrakte Klassen. Das Wort *Interface* bedeutet *Schnittstelle*. Somit kann auch ein Interface als eine Art Schnittstelle innerhalb einer Programmiersprache gesehen werden. Tatsächlich handelt es sich dabei um die Definition verschiedener Methoden, die aber keine Implementation, also funktionellen Code beinhalten. Dieser wird stattdessen von der Klasse zur Verfügung gestellt, die das Interface implementiert.

***Vereinheitlichung***

Durch die Vereinheitlichung der Methoden, die ja bereits im Interface deklariert wurden, ergibt sich auch eine einheitliche Behandlung der entsprechenden Funktionalität aller Klassen, die das Interface implementieren. Der eigentliche Code innerhalb der Methoden kann unterschiedlich sein, angesprochen werden sie aber alle auf

dem gleichen Weg mit den gleichen Parametern, und auch der Rückgabewert ist einheitlich.

Vermutlich werden Sie sich jetzt fragen, warum denn statt der Interfaces keine abstrakte Klassen verwendet werden. Immerhin ist es auch dort so, dass die Methoden erst in der abgeleiteten Klasse implementiert werden (genau wie bei Interfaces) und dass sie abgeleitet werden müssen. Es gibt jedoch einen gravierenden Unterschied, denn Interfaces ermöglichen Mehrfachvererbung.

***Mehrfachvererbung***

Es ist somit möglich, mehrere Interfaces zu implementieren, und nicht nur eines, wie es mit den Klassen der Fall ist. Im Prinzip handelt es sich bei der Verwendung eines Interface auch um eine Art der Vererbung, denn die Klasse, die das Interface implementiert, erbt ja alle darin deklarierten Methoden (auch wenn die Funktionalität noch bereitgestellt werden muss). Allerdings ist es auch möglich, mehrere Interfaces gleichzeitig in einer Klasse zu verwenden, bei der Ableitung von einer abstrakten Klasse wäre dies nicht möglich.

Das folgende Beispiel finden Sie komplett auf der CD im Verzeichnis BEISPIELE\KAPITEL_8\INTERFACES. Es wurde allerdings nur eine Main-Methode implementiert. Sie können die beiden anderen natürlich auch testen.

### 8.2.1 Deklaration eines Interface

Die Deklaration eines Interface beschränkt sich auf die Angabe der Methodenköpfe ohne Rumpf oder Implementationsteil. Die Funktionalität wird später in der Klasse zur Verfügung gestellt. Das Beispiel zeigt ein Interface für Geometrische Berechnungen. Übergabeparameter benötigen wir nicht, da alle relevanten Informationen bereits in unserem Objekt gespeichert sind und wir diese natürlich verwenden können. Lediglich ein Rückgabewert vom Typ **double** wird implementiert, denn irgendwie müssen die Werte ja zurückgeliefert werden.

```
/* Beispiel Interface 1 */
/* Autor:   Frank Eller */
/* Sprache: C#          */

using System;
interface IGeometric
{
  double GetArea();
  double GetPerimeter();
}
```

Das deklarierte Interface ist sehr einfach, implementiert nur je eine Methode für die Berechnung der Fläche und des Umfangs. Für verschiedene geometrische Objekte wie z.B. Kreis, Dreieck oder Rechteck müssen diese Berechnungen aber auf unterschiedliche Art erfolgen. Dennoch können wir mit Hilfe eines Interface diese Informationen auf die gleiche Art ermitteln.

Modifikatoren sind innerhalb der Interfaces nicht erlaubt, bzw. der Compiler meldet sich mit einer Fehlermeldung, wenn Sie Modifikatoren wie z.B. `public` verwenden wollen. Die Methoden des Interface werden später in der Klasse implementiert.

### 8.2.2 Deklaration der geometrischen Klassen

Zunächst deklarieren wir eine Basisklasse mit Methoden und Variablen, die für jedes unserer geometrischen Objekte relevant sind. In unserem Fall, um es ein wenig einfacher zu machen, lediglich eine Methode zum Zeichnen und eine Methode zur Ausgabe an den Drucker.

```
/* Beispiel Geometrie: Basisklasse */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

public class GeoClass
{
  public GeoClass()
  {
    //Initialisierungsroutinen
  }

  public virtual void DrawScreen()
  {
    //Code zum Zeichnen auf dem Bildschirm
  }

  public virtual void DrawPrinter()
  {
    //Code zum Zeichnen auf dem Drucker
  }
}
```

Fertig ist unsere Basisklasse, von der wir unsere Objekte nun ableiten können. Die Syntax unserer Klassendeklaration, die wir ja bereits einmal erweitert haben, muss nun nochmals erweitert werden.

```
[Modifikator] Ergebnistyp Bezeichner : Basis [,Interfaces]
```

Für unser Beispiel wollen wir nun Klassen zur Verfügung stellen, die jeweils einen Kreis, ein Rechteck und ein Quadrat repräsentieren. Der Code für die Ausgabe wird nicht implementiert, in diesem Fall wird einfach der Code der Basisklasse aufgerufen. Die Methoden für die Berechnungen müssen wir allerdings implementieren.

```
/* Beispiel Geometrie: Klassen ableiten */
/* Autor:   Frank Eller                 */
/* Sprache: C#                          */

public class Rechteck : GeoClass, IGeometric
{
  public double SeiteA = 0;
  public double SeiteB = 0;

  public Rechteck()
  {
    //Standard-Konstruktor
  }

  public Rechteck(double SeiteA, double SeiteB)
  {
    this.SeiteA = SeiteA;
    this.SeiteB = SeiteB;
  }

  //Implementation der Interface-Methoden
  public double GetArea()
  {
    if ((SeiteA != 0)&&(SeiteB != 0))
      return (SeiteA*SeiteB);
    else
      return -1;
  }

  public double GetPerimeter()
  {
    if ((SeiteA != 0)&&(SeiteB != 0))
      return (SeiteA*2+SeiteB*2);
    else
      return -1;
  }
}
```

```
public class Kreis : GeoClass, IGeometric
{
  public double Radius = 0;

  public Kreis()
  {
    //Standard-Konstruktor
  }

  public Kreis(double Radius)
  {
    this.Radius = Radius;
  }

  //Implementation der Interface-Methoden
  public double GetArea()
  {
    if (Radius != 0)
    {
      double d = Radius*2;
      return (((Math.Pow(d,2)*Math.PI)/4));
    }
    else
      return -1;
  }

  public double GetPerimeter()
  {
    if (Radius != 0)
      return (2*Radius*Math.PI);
    else
      return -1;
  }
}

public class Quadrat : GeoClass, IGeometric
{
  public double SeiteA = 0;

  public Quadrat()
  {
    //Standard-Konstruktor
  }
  public Quadrat(double SeiteA)
  {
    this.SeiteA = SeiteA;
```

```
  }

  //Implementation der Interface-Methoden
  public double GetArea()
  {
    if (SeiteA != 0)
      return (Math.Pow(SeiteA,2));
    else
      return -1;
  }

  public double GetPerimeter()
  {
    if (SeiteA != 0)
      return (SeiteA*4);
    else
      return -1;
  }
}
```

Damit wären die abgeleiteten Klassen fertig erstellt. Wir können diese jetzt wie gewohnt in unserem Programm verwenden. Um aber den Sinn und Zweck von Interfaces herauszustellen, wollen wir auf eine bestimmte Art vorgehen.

### 8.2.3 Das Interface verwenden

Wir wissen, dass jedes geometrische Objekt von der Basisklasse `GeoClass` abgeleitet ist, durch die Polymorphie also auch ein Objekt vom Typ dieser Klasse ist. Die folgende Deklaration ist demnach möglich:

```
/* Beispiel Geometrie: Main()-Funktion 1 */
/* Autor:   Frank Eller                  */
/* Sprache: C#                           */

public class TestClass
{
  public static void Main()
  {
    GeoClass[] geo = new GeoClass[3];
    geo[0] = new Rechteck(10,20);

    geo[1] = new Kreis(30);
    geo[2] = new Quadrat(20);
  }
}
```

Über eine foreach-Schleife können wir nun die Objekte nacheinander durchgehen und, obwohl es sich um verschiedene geometrische Darstellungen handelt, stets die gleiche Art der Abfrage durchführen.

```
/* Beispiel Geometrie: Main()-Funktion 2 */
/* Autor:   Frank Eller                  */
/* Sprache: C#                           */

public class TestClass
{
  public static void Main()
  {
    GeoClass[] geo = new GeoClass[3];
    geo[0] = new Rechteck(10,20);
    geo[1] = new Kreis(30);
    geo[2] = new Quadrat(20);

    foreach (GeoClass g in geo)
    {
      IGeometric x = (IGeometric)(g);
      Console.WriteLine(x.GetArea());
      Console.WriteLine(x.GetPerimeter());
    }
  }
}
```

Das Casting in den Typ IGeometric ist unbedingt notwendig, da Sie auf die Methoden dieses Datentyps zugreifen. Zwar ist die eigentliche Funktionalität in der jeweiligen Klasse programmiert, der Zugriff funktioniert aber dennoch nur über IGeometric.

Um dieses Verhalten zu erklären greifen wir zurück auf den Datentyp object, der als Basisklasse aller Klassen auch jeden beliebigen Datentyp enthalten kann. Durch Casting kann der Datentyp objekt in den Datentyp konvertiert werden, den er enthält.

Ebenso funktioniert es bei den Klassen. Der Grund ist die Polymorphie, wenn also, um bei unserem Beispiel zu bleiben, ein Objekt des ursprünglichen Datentyps GeoClass erzeugt wird, kann es sich dabei um ein Objekt des Typs Kreis, Rechteck oder Quadrat handeln, da GeoClass die Basisklasse dieser Objekte ist.

Gleichzeitig ist es möglich, da ja auch ein Interface in den Klassen enthalten ist, durch Casting auch in den Datentyp des Interface zu konvertieren. Und bei mehreren Interfaces ist dies natürlich auch mehrfach möglich.

Ein solches Vorgehen, wie wir es im obigen Beispiel gesehen haben, ist aber nur möglich, wenn das Interface in der Klasse auch implementiert wird. Wenn wir nun mit verschiedenen geometrischen Objekten arbeiten, die alle von unserer Basisklasse `GeoClass` abgeleitet sind, wissen wir nicht, ob alle diese Objekte das Interface `IGeometric` implementieren. Wir benötigen also eine Möglichkeit, dies zu erkennen.

Eigentlich ist das aber nicht besonders schwierig, denn wie bereits weiter oben erklärt wurde, handelt es sich bei der Implementierung eines Interface eigentlich auch nur um eine Vererbung. Unser Objekt ist also einerseits grundsätzlich vom Typ `GeoClass`, da es von diesem abgeleitet ist. Es ist aber auch vom Typ `IGeometric`, denn auch davon ist es abgeleitet, was möglich ist, weil es sich bei `IGeometric` um ein Interface handelt.

*is*

Wir haben bereits oben gesehen, dass das Casting möglich ist. Damit ist es natürlich auch möglich, die Kontrolle durchzuführen. Wir überprüfen einfach, ob unser Objekt vom Typ `IGeometric` ist, und wenn ja, können wir die darin enthaltenen Methoden nutzen. Für die Kontrolle verwenden wir das reservierte Wort **is**, mit dem wir prüfen können, ob ein Objekt von einem bestimmten Datentyp abstammt.

```
/* Beispiel Geometrie: Main()-Funktion 3 */
/* Autor:   Frank Eller                  */
/* Sprache: C#                           */

public class TestClass
{
  public static void Main()
  {
    GeoClass[] geo = new GeoClass[3];
    geo[0] = new Rechteck(10,20);
    geo[1] = new Kreis(30);
    geo[2] = new Quadrat(20);

    foreach (GeoClass g in geo)
    {
      if (g is IGeometric)
      {
        IGeometric x = (IGeometric)(g);
        Console.WriteLine(x.GetArea());
        Console.WriteLine(x.GetPerimeter());
      }
      else
        Console.WriteLine("Interface nicht implementiert");
    }
  }
}
```

*as*

Eine weitere Möglichkeit der Kontrolle ist über das reservierte Wort **as** möglich, das ebenfalls eine Konvertierung durchführt.

```
/* Beispiel Geometrie: Main()-Funktion 4 */
/* Autor:   Frank Eller                  */
/* Sprache: C#                           */

public class TestClass
{
  public static void Main()
  {
    GeoClass[] geo = new GeoClass[3];
    geo[0] = new Rechteck(10,20);
    geo[1] = new Kreis(30);
    geo[2] = new Quadrat(20);

    foreach (GeoClass g in geo)
    {
      IGeometric x = g as IGeometric;
      if (x != null)
      {
        Console.WriteLine(x.GetArea());
        Console.WriteLine(x.GetPerimeter());
      }
      else
        Console.WriteLine("Interface nicht implementiert");
    }
  }
}
```

**as** bewirkt eine Umwandlung in den angegebenen Datentyp. In unserem Fall in den Typ des Interface. Falls das Interface nicht implementiert ist, ergibt sich als Wert für die Variable **null**, sie referenziert also in diesem Moment kein Objekt. Daher können wir mit einer einfachen **if**-Abfrage eine Kontrolle vornehmen.

### 8.2.4 Mehrere Interfaces verwenden

Als Beispiel werden wir wieder eine unserer bekannten Klassen nehmen, diesmal werden wir aber ein weiteres Interface hinzufügen, das den Namen der Klasse ausspuckt. Das Beispiel zeigt, wie einfach es ist, mehrere Interfaces zu verwenden. In diesem Fall wird das neu hinzugefügte Interface einfach durch Komma getrennt wieder hinter die Deklaration geschrieben. Damit haben wir den Fall der Mehrfachvererbung, denn jetzt sind in einer Klasse zwei unterschiedliche Interfaces implementiert. Der folgende Quelltext zeigt das komplette Programm.

```
/* Beispiel Mehrfachvererbung */
/* Autor:   Frank Eller       */
/* Sprache: C#               */

using System;

public class GeoClass
{
  public GeoClass()
  {
    //Initialisierungsroutinen
  }

  public virtual void DrawScreen()
  {
    //Code zum Zeichnen auf dem Bildschirm
  }

  public virtual void DrawPrinter()
  {
    //Code zum Zeichnen auf dem Drucker
  }
}

interface IGeometric
{
  double GetArea();
  double GetPerimeter();
}

interface IName
{
  string ReturnName();
}

public class Rechteck : GeoClass, IGeometric
{
  public double SeiteA = 0;
  public double SeiteB = 0;

  public Rechteck()
  {
    //Standard-Konstruktor
  }
```

```
  public Rechteck(double SeiteA, double SeiteB)
  {
    this.SeiteA = SeiteA;
    this.SeiteB = SeiteB;
  }

  //Implementation der Interface-Methoden
  public double GetArea()
  {
    if ((SeiteA != 0)&&(SeiteB != 0))
      return (SeiteA*SeiteB);
    else
      return -1;
  }

  public double GetPerimeter()
  {
    if ((SeiteA != 0)&&(SeiteB != 0))
      return (SeiteA*2+SeiteB*2);
    else
      return -1;
  }
}

public class Kreis : GeoClass, IGeometric
{
  public double Radius = 0;

  public Kreis()
  {
    //Standard-Konstruktor
  }

  public Kreis(double Radius)
  {
    this.Radius = Radius;
  }

  //Implementation der Interface-Methoden
  public double GetArea()
  {
    if (Radius != 0)
    {
      double d = Radius*2;
      return (((Math.Pow(d,2)*Math.PI)/4));
    }
```

```
    else
      return -1;
  }

  public double GetPerimeter()
  {
    if (Radius != 0)
      return (2*Radius*Math.PI);
    else
      return -1;
  }
}

public class Quadrat : GeoClass, IGeometric, IName
{
  public double SeiteA = 0;
  string theName = "Quadrat";

  public Quadrat()
  {
    //Standard-Konstruktor
  }

  public Quadrat(double SeiteA)
  {
    this.SeiteA = SeiteA;
  }

  //Implementation der Interface-Methoden
  public double GetArea()
  {
    if (SeiteA != 0)
      return (Math.Pow(SeiteA,2));
    else
      return -1;
  }
  public double GetPerimeter()
  {
    if (SeiteA != 0)
      return (SeiteA*4);
    else
      return -1;
  }
```

```
  public string ReturnName()
  {
    return theName;
  }
}

public class TestClass
{
  public static void Main()
  {
    GeoClass[] geo = new GeoClass[3];
    geo[0] = new Rechteck(10,20);
    geo[1] = new Kreis(30);
    geo[2] = new Quadrat(20);

    foreach (GeoClass g in geo)
    {
      if (g is IName)
      {
        IName x = (IName)(g);
        Console.WriteLine(x.ReturnName());
      }
    }
  }
}
```

Das komplette Beispielprogramm finden Sie auf der beiligenden CD im Verzeichnis BEISPIELE\KAPITEL_8\INTERFACES2.

Die Kontrolle des Interface funktioniert natürlich wie vorher. Durch das Casting können wir der Variablen als Datentyp jedes der implementierten Interfaces zuweisen. Damit ist auch der Zugriff auf alle im Interface enthaltenen Methoden möglich.

### 8.2.5 Explizite Interface-Implementierung

Oftmals wird es der Fall sein, dass Ihnen Fremdanbieter Komponenten für die Programmierung zur Verfügung stellen (oder Sie sie käuflich erwerben) und damit auch entsprechende Interfaces, die Sie natürlich in Ihren eigenen Applikationen ebenfalls verwenden können. Allerdings können Sie sie nicht ändern. Was geschieht also, wenn Sie bei der Deklaration einer Klasse zwei Interfaces verwenden, die beide eine Funktion besitzen, die den gleichen Namen hat? Sehen Sie sich das folgende Beispiel an.

```
/* Beispiel explizite Interface-Implementierung 1 */
/* Autor:   Frank Eller                           */
/* Sprache: C#                                    */

interface Interface1
{
  void doAusgabe();
}

interface Interface2
{
  void doAusgabe()
}

class TestClass : Interface1, Interface2
{
  public void doAusgabe()
  {
    //Frage:
    //Welche Interface-Methode wird benutzt?
  }
}
```

Bei diesem Beispiel hätten Sie schlechte Karten, denn der Compiler könnte nicht feststellen, welches der beiden Interfaces gemeint ist. Damit bleibt für ihn nur der Ausweg, aufzugeben und eine Fehlermeldung auszugeben.

***Interface qualifizieren***

Es wäre mehr als unbefriedigend, wenn Sie wegen einer solchen Namenskollision eines der Interfaces weglassen müssten. Sie wären mit Recht verärgert darüber. Allerdings bietet C# auch hier einen Ausweg, indem es Ihnen ermöglicht, in der Deklaration anzugeben, welches Interface benutzt werden soll. Dazu schreiben Sie einfach den Bezeichner des Interface vor den Methodenbezeichner und trennen beide mit einem Punkt. Das Interface bzw. die Methode wird also qualifiziert.

```
/* Beispiel explizite Interface-Implementierung 2 */
/* Autor:   Frank Eller                           */
/* Sprache: C#                                    */

interface Interface1
{
  void doAusgabe();
}

interface Interface2
```

```
{
  void doAusgabe()
}

class TestClass : Interface1, Interface2
{
  public void Interface1.doAusgabe()
  {
    Console.WriteLine("Interface 1 benutzt");
  }

  public void Interface2.doAusgabe()
  {
    Console.WriteLine("Interface 2 benutzt");
  }
}
```

Diese Art der Implementierung von Interface-Methoden nennt man auch *explizite Interface-Implementierung*. Indem Sie das Interface mit angeben vermeiden Sie, dass es zu Namenskonflikten bei der Verwendung mehrerer Interfaces in derselben Klasse kommt.

## 8.3 Delegates

Kommen wir nun zu einem anderen Datentyp, der deshalb wichtig ist, weil C# keine Zeiger kennt. Es wäre also nützlich, wenn es eine Möglichkeit gäbe, einen Zeiger auf eine Methode zu erzeugen und somit genau diese Methode aufrufen zu können; so hätte man eine sinnvolle und mächtige Erweiterung der Sprache. Auch wenn Ihnen das im Moment noch nicht so klar ist, im weiteren Verlauf des Kapitels werden Sie sehen, wie es funktioniert. Denn selbstverständlich ist es möglich, wenn auch nicht über einen direkten Zeiger, sondern über einen so genannten *Delegate*, einen „Abgesandten".

**Forward-Deklaration**

Ein Delegate funktioniert ähnlich wie ein Zeiger auf eine Funktion. Deklariert wird nur der Kopf der späteren Funktion, implementiert wird sie in einer Klasse. Das ist ein Moment, wo es eine Abweichung von der Regel gibt. Es wurde am Anfang behauptet, C# benötige keine Forward-Deklarationen. Eigentlich tun wir mit Delegates aber nichts anderes, wir deklarieren eine Funktion, die später implementiert wird. Es ist dennoch nicht ganz das Gleiche, denn Delegates können auch innerhalb von Klassen deklariert werden. Da es jetzt ein wenig kompliziert werden kann, gehen wir Schritt für Schritt vor und bauen uns langsam ein Beispiel auf, mit dem wir arbeiten können.

### 8.3.1 Deklaration eines Delegate

Zunächst werden wir unseren Delegate deklarieren. Dabei handelt es sich – wie bereits gesagt – um den Prototypen einer Funktion, der später dazu benutzt wird, eine beliebige Funktion mit den gleichen Parametern aufzurufen.

```
public delegate bool bigger(object a, object b);
```

Die spätere Funktion soll dazu dienen, zwei Adressen zu vergleichen. Allerdings soll dieser Vergleich auf unterschiedlichen Vorgaben basieren – einmal soll nach der Postleitzahl verglichen werden, einmal nach dem Namen. Damit ist der Sinn eines Delegate bereits ausreichend erklärt, denn es wird nur noch ein Methodenaufruf benötigt, das Verhalten der Klasse kann jedoch immer ein anderes sein.

Für die Daten können wir entweder eine Klasse oder einen **struct** verwenden. In diesem Fall habe ich mich für den **struct** entschieden, da ohnehin nur Daten enthalten sind. Die Deklaration sieht folgendermaßen aus:

```
public struct myAddress

{
  public string name;

  public string strasse;

  public int plz;

  public string ort;

}
```

Somit haben wir den Delegate und den **struct** für unsere Daten. Jetzt können wir beginnen, die Klasse aufzubauen, die die eigentliche Sortierung bzw. die gesamte Arbeit mit den Daten ausführen soll.

### 8.3.2 Deklaration einer Klasse

Im folgenden Listing sehen Sie die Grundstruktur der Klasse, die wir dann mit Funktionalität füllen wollen.

```
/* Beispiel Delegates: Basisklasse */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

public class Sorter
{
  private myAddress[] adr = new myAddress[5];

  void Swap(ref myAddress a, ref myAddress b)
  {

  }

  public void doSort(bigger isBigger)
  {

  }

  public void SetAddr(int a,string n,string s,
                      string o, int p)
  {

  }

  public string adrToString(int a)
  {

  }

  public static bool plzBigger(object a, object b)
  {

  }

  public static bool nameBigger(object a, object b)
  {

  }
}
```

Die beiden letzten Methoden, `plzBigger()` und `nameBigger()`, werden später dazu dienen, jeweils zwei Elemente des Array `adr[]` zu vergleichen. Den deklarierten Delegate werden wir dazu verwenden, jeweils die richtige Methode aufzurufen. Somit wird das Array einmal nach der Postleitzahl und einmal nach dem Namen kontrolliert.

### 8.3.3 Die Methoden der Klasse

Kommen wir zur Implementation der Funktionalität unserer Klasse. Die einfachste Methode ist die Methode `Swap()`, in der lediglich zwei Elemente des gleichen Typs vertauscht werden. Die Elemente wurden als Referenzparameter übergeben, dadurch haben wir also automatisch die vertauschte Reihenfolge.

```
/* Beispiel Delegates: Methoden 1 */
/* Autor:   Frank Eller           */
/* Sprache: C#                    */

void Swap(ref myAddress a, ref myAddress b)
{
  myAddress x = new myAddress();
  x = a;
  a = b;
  b = x;
}
```

Das Vertauschen von Werten funktioniert auch mit einem **struct**, alle Werte werden einfach zusammen ausgetauscht. Ebenso einfach sind die Methoden `SetAddr()` und `AddrToString()`, die dazu dienen, den Inhalt eines Adressenelements festzulegen bzw. zwecks Ausgabe zurückzuliefern.

Wenn Sie zusammengesetzte Datentypen benutzen, wie z.B. **structs**, können Sie die Daten dennoch komplett mittels einer einfachen Zuweisung kopieren. Wenn `a` und `b` structs des gleichen Typs sind, weist der Befehl `a=b` dem **struct** `a` alle enthaltenen Werte des **struct** `b` zu.

```
/* Beispiel Delegates: Methoden 2 */
/* Autor:   Frank Eller           */
/* Sprache: C#                    */

public void SetAddr(int a,string n,string s,string o,int p)
{
  this.adr[a].name    = n;
  this.adr[a].strasse = s;
  this.adr[a].plz     = p;
  this.adr[a].ort     = o;
}

public string adrToString(int a)
{
  return string.Format("{0},{1}",this.adr[a].name,this.adr[a].plz);
}
```

Bis jetzt sind wir immer noch im trivialen Teil. Auch die Routine, die den eigentlichen Sortiervorgang durchführt, ist im Prinzip bereits bekannt, denn wir haben im Verlauf des Buchs schon einmal Werte sortiert. Wir benutzen die gleiche Routine, allerdings übergeben wir ihr eine Instanz unseres Delegate, der auf die Funktion verweisen wird, die die eigentliche Kontrolle durchführt.

```
/* Beispiel Delegates: Methoden 3 */
/* Autor:   Frank Eller           */
/* Sprache: C#                    */

public void doSort(bigger isBigger)
{
  bool changed = false;
  do
  {
    changed = false;
    for (int i=1;i<5;i++)
    {
      if (isBigger(adr[i-1],adr[i]))
      {
        Swap(ref adr[i-1],ref adr[i]);
        changed = true;
      }
    }
  } while (changed);
}
```

Die zwei Routinen, die die Kontrolle durchführen, sind wiederum recht einfach gehalten. Einmal wird überprüft, welcher Name der größere ist (also im Alphabet weiter hinten steht), und einmal werden die Postleitzahlen kontrolliert. Damit die Funktion unabhängig von Datentypen arbeiten kann, wurden, wie schon an der Deklaration des Delegate zu sehen ist, als Parameter Werte vom Typ **object** verwendet. Somit können alle Datentypen übergeben werden, C# kümmert sich automatisch um die Konvertierung in den Datentyp **object** mittels Boxing. Außerdem sind beide Methoden als **static** deklariert, damit wir sie instanzenunabhängig verwenden können.

```
/* Beispiel Delegates: Methoden 4 */
/* Autor:   Frank Eller           */
/* Sprache: C#                    */

public static bool plzBigger(object a, object b)
{
  myAddress x = (myAddress)(a);
  myAddress y = (myAddress)(b);
```

```
  return (x.plz>y.plz)?true:false;
}

public static bool nameBigger(object a, object b)
{
  myAddress x = (myAddress)(a);
  myAddress y = (myAddress)(b);
  return (string.Compare(x.name,y.name)>0);
}
```

So, damit wäre die Klasse fertig. Alles, was nun noch zu tun ist, ist eine Instanz des Delegate zu erzeugen, wobei die zu benutzende statische Methode angegeben wird. Das tun wir natürlich im Hauptprogramm der Anwendung.

### 8.3.4 Das Hauptprogramm

Bevor wir mittels unseres Delegate sortieren, müssen wir zunächst ein Array aufbauen und entsprechend Werte einlesen, denn ohne Werte kann auch nicht sortiert werden. Dann werden wir den Delegate auf die zu verwendende Methode umbiegen, indem wir eine neue Instanz desselben unter Angabe des Methodennamens als Übergabeparameter erzeugen.

```
/* Beispiel Delegates: Hauptprogramm */
/* Autor:   Frank Eller              */
/* Sprache: C#                       */

public class MainClass
{
  public static int Main(string[] args)
  {
    Sorter mySort = new Sorter();

    //Einlesen der Werte
    for (int i=0;i<5;i++)
    {
      Console.Write("Name: ");
      string n = Console.ReadLine();
      Console.Write("Strasse: ");
      string s = Console.ReadLine();
      Console.Write("PLZ: ");
      int p = Console.ReadLine().ToInt32();
      Console.Write("Ort: ");
      string o = Console.ReadLine();
```

```
      mySort.SetAddr(i,n,s,o,p);
    }

    //Delegate instanziieren
    bigger istBigger = new bigger(Sorter.plzBigger);

    //Aufrufen der Sortiermethode
    mySort.doSort(istBigger);

    //Ausgabe des sortierten Array
    for (int i=0;i<5;i++)
    {
      Console.WriteLine(mySort.adrToString(i));
    }

    //Und das Ganze noch mal mit dem Namen
    istBigger = new bigger(Sorter.nameBigger);
    mySort.doSort(istBigger);
    Console.WriteLine();
    for (int i=0;i<5;i++)
    {
      Console.WriteLine(mySort.adrToString(i));
    }

    return 0;
  }
}
```

Das Delegate-Objekt `istBigger` wird unter Angabe der zu verwendenden Methode instanziiert. Hier wird also die Entscheidung getroffen, welche Art der Sortierung vorgenommen werden soll. In der Methode `doSort()` unserer Sortierungsklasse wird dann automatisch bei Aufruf des als Parameter übergebenen Delegate die richtige Methode ausgewählt.

Delegates werden hauptsächlich zur Definition von Ereignissen verwendet, die eine Klasse auslösen kann. Da es sich hierbei aber um etwas handelt, was auch vom Betriebssystem ausgeht (denn auch dies benutzt Ereignisse bzw. löst eigene Ereignisse aus), muss eine gemeinsame Form der Deklaration gefunden werden, um kompatibel zu bleiben. Genau dazu sind Delegates optimal geeignet.

Sie finden das komplette Beispielprogramm wie gehabt auf der CD im Verzeichnis BEISPIELE\KAPITEL_8\DELEGATES.

## 8.4 Zusammenfassung

In diesem Kapitel sind wir ein wenig tiefer in die Möglichkeiten eingestiegen, die eine objektorientierte Programmiersprache bietet. Es ging vor allem um Vererbung, um die verschiedenen Arten der Deklaration einer Klasse und die Möglichkeiten, die Interfaces und Delegates bieten. Somit haben Sie zwar eine weitere Möglichkeit der Untergliederung eines Programms, müssen aber darauf achten, dass es letzten Endes nicht zu kompliziert wird. Sie sehen an diesen Möglichkeiten, dass vor allem bei der Planung größerer Projekte sehr sorgfältig vorgegangen werden sollte.

## 8.5 Kontrollfragen

1. Von welchen Klassen muss in jedem Fall abgeleitet werden?
2. Mit welchem Modifikator werden Methoden deklariert, die von einer abgeleiteten Klasse überschrieben werden können?
3. Wozu dient der Modifikator `override`?
4. Was ist die Eigenschaft einer versiegelten Klasse?
5. Woran kann man eine versiegelte Klasse erkennen?
6. Was ist der Unterschied zwischen abstrakten Klassen und Interfaces, von denen ja in jedem Fall abgeleitet werden muss?
7. Kann ein Interface Funktionalität enthalten?
8. Wie können Methoden gleichen Namens in unterschiedlichen Interfaces dennoch verwendet werden?
9. Wie kann auf die Methoden eines Interface zugegriffen werden, das in einer abgeleiteten Klasse implementiert wurde?
10. Was bedeutet das Wort `delegate`?
11. Wozu dienen Delegates?
12. Was ist die Entsprechung eines Delegate in anderen Programmiersprachen?

## 8.6 Übungen

An dieser Stelle auch wieder einige Übungen zur Vertiefung des Stoffes.

### Übung 1

Erstellen Sie eine Basisklasse. Die Klasse soll Personen aufnehmen können, mit Name und Vorname.

### Übung 2

Leiten Sie zwei Klassen von der Basisklasse ab, eine für männliche Personen, eine für weibliche Personen. Implementieren Sie für die neuen Klassen eine Zählvariable, mit der Sie die Anzahl Männer bzw. Frauen erfassen können.

### Übung 3

Die Ausgabe des Namens bzw. des Geschlechts soll über ein Interface realisiert werden. Erstellen Sie ein entsprechendes Interface und binden Sie es in die beiden neuen Klassen ein. Sorgen Sie dafür, dass sich die Ausgaben später wirklich unterscheiden, damit eine Kontrolle möglich ist.

# 9 Eigenschaften und Ereignisse

Klassen haben wir bereits in Kapitel 3, das die Strukturierung eines C#-Programms behandelt, besprochen. Tatsächlich ist jedes C#-Programm so aufgebaut, dass einzelne Klassen miteinander interagieren und so die Gesamtfunktionalität eines Programms bestimmen. Wir haben aber auch bereits einige Klassen besprochen, die vom .net-Framework zur Verfügung gestellt werden. In diesem Kapitel werden wir unser Wissen über Klassen erweitern.

Im letzten Kapitel haben wir einiges über Vererbung gelernt, weiterhin über Interfaces und Delegates. Letztere werden wir später nochmals benötigen, wenn es um die Deklaration und das Auslösen von Ereignissen geht. Doch vorher wollen wir uns mit der Erweiterung der Zugriffsmöglichkeiten auf die in einer Klasse deklarierten Variablen beschäftigen, den so genannten *Eigenschaften* oder *Properties*.

## 9.1 Eigenschaften

Bisher haben wir in unseren Klassen Felder und Methoden zur Verfügung gestellt. Feldern konnten wir Werte zuweisen, Methoden konnten wir innerhalb unserer Programme aufrufen und ihre Funktionalität nutzen. Weitere Möglichkeiten, die eine Klasse bietet, sind so genannte Eigenschaften (*Properties*) und Ereignisse (*Events*). Eigenschaften bieten eine spezielle Art des Zugriffs auf die Werte eines Felds, während Ereignisse dazu dienen, anderen Klassen bzw. auch Windows mitzuteilen, dass ein bestimmter Vorgang stattgefunden hat. Kümmern wir uns in diesem Zusammenhang zunächst um die Eigenschaften, die eine Klasse zur Verfügung stellen kann.

***Eigenschaften und Felder***

Für den Programmierer einer Anwendung verhalten sich Eigenschaften eigentlich wie Felder einer Klasse, man kann ihnen Werte zuweisen oder auch den darin enthaltenen Wert auslesen. Der Unterschied zu einem herkömmlichen Feld besteht darin, dass Eigenschaften für

das Auslesen bzw. die Wertzuweisung Methoden benutzen, die automatisch aufgerufen werden, sobald entweder eine Zuweisung oder das Auslesen eines Wertes gefordert ist. Der Vorteil dieser Vorgehensweise ist, dass nun die Deklaration der Klasse vollständig von der Funktionalität getrennt ist. Das bedeutet, wenn eine Änderung bezüglich des Zugriffs auf die entsprechende Eigenschaft notwendig ist, müssen Sie nur die Implementation der Klasse ändern und nicht jedes Vorkommen der Zuweisung an das entsprechende Feld.

Der Zugriff auf die Werte einer Eigenschaft erfolgt über Zugriffsmethoden, den *Getter* und den *Setter*. Die eine Methode liefert den enthaltenen Wert zurück, die andere setzt ihn. Ich habe hierbei absichtlich die Originalbegriffe belassen, da die verwendeten Methoden ebenfalls die Namen **get** und **set** besitzen müssen.

### 9.1.1 Eine Beispielklasse

An einem Beispiel wollen wir den Unterschied zwischen einem herkömmlichen Feld und einer Eigenschaft deutlich machen:

```
/* Beispielklasse Eigenschaften 1 */
/* Autor:   Frank Eller           */
/* Sprache: C#                    */

public class cArtikel
{
  public double Price = 0;
  public string Name  = "";

  public cArtikel(string Name, double Price)
  {
    this.Price = Price;
    this.Name  = Name;
  }
}
```

Die obige Klasse dient dazu, einen Warenartikel mit Namen und Preis abzuspeichern. Dazu wird wie immer eine Instanz der Klasse erzeugt, wobei Name und Preis des Artikels direkt übergeben werden können. Nach Erzeugen der Instanz können Preis und Bezeichnung über die öffentlichen Felder `Price` und `Name` geändert werden.

Implementiert mit Eigenschaften sieht die Klasse dann folgendermaßen aus:

```
/* Beispielklasse Eigenschaften 2 */
/* Autor:   Frank Eller           */
/* Sprache: C#                    */

using System;

public class cArtikel
{
  private double Price = 0;
  private string Name  = "";

  public string theName
  {
    get
    {
      return Name;
    }
    set
    {
      Name = value;
    }
  }

  public double thePrice
  {
    get
    {
      return Price;
    }
    set
    {
      Price = value;
    }
  }

  public cArtikel(string Name, double Price)
  {
    this.Price = Price;
    this.Name  = Name;
  }
}
```

Der Unterschied liegt in den Methoden **get** und **set** und natürlich darin, dass auf den Inhalt der Variablen `Price` und `Name` jetzt über die

Eigenschaften thePrice und theName zugegriffen wird. Der zunächst sichtbare Nachteil, dass mehr Code zu schreiben ist, relativiert sich durch die Tatsache, dass bei einer Änderung der Zuweisung nur noch die Klasse geändert werden muss, nicht aber jede Zuweisung im Programm. Die Klasse im Einsatz finden Sie auf der CD im Verzeichnis BEISPIELE\KAPITEL_9\EIGENSCHAFTEN1, natürlich mit einem entsprechenden Hauptprogramm.

***get und set***

Die Methoden **get** und **set** sind festgelegt, die Namen können Sie also nicht ändern. Wohl aber die Zugriffsmöglichkeit auf die Eigenschaft. So ist es z.B. auch möglich, eine Eigenschaft zu erstellen, die nur innerhalb der Klasse gesetzt werden kann (denn dann kann ja auch auf die entsprechende Variable zugegriffen werden), aber ansonsten nur einen Lesezugriff zur Verfügung stellt. Dazu verzichten Sie einfach auf die **set**-Methode, schon kann der Anwender Ihrer Klasse den Wert von außerhalb nicht mehr ändern.

***value***

Der in der **set**-Methode verwendete Wert **value** ist ein festgelegter Wert – er heißt immer so, für alle Eigenschaften, die deklariert werden. Die Eigenschaft selbst hat ja einen Datentyp, **value** ist von diesem Datentyp und enthält bei einer Zuweisung stets den zuzuweisenden Wert. Auch ein Indiz dafür ist, dass es sich bei **value** um ein reserviertes Wort mit festgelegter Bedeutung handelt.

Die Namen der Methoden **get** und **set** sowie der Wert **value** sind vorgegeben und können nicht geändert werden. Sie können aber mit **value** wie mit jedem anderen Wert arbeiten. **value** ist dabei immer vom gleichen Datentyp wie die Eigenschaft, für die er benutzt wird.

### 9.1.2 Die Erweiterung des Beispiels

Nehmen wir an, die von uns auf diese Art erstellte Klasse würde in einem anderen Programm genutzt. Nun soll gewährleistet werden, dass der Preis sowohl in Euro als auch in DM zur Verfügung steht, wobei sich an der Zuweisung nichts ändern soll. Für diesen Fall bauen wir die Klasse nun einfach mal um, wir fügen also ein weiteres Feld hinzu (den Euro-Preis) und legen fest, auf welche Art die Zuweisung geschehen soll – in DM oder in Euro. Außerdem legen wir eine Konstante für den Umrechnungskurs an. Wozu das alles dient, werden wir dann im weiteren Verlauf sehen. Zunächst hier die neue Klasse:

```
/* Beispielklasse Eigenschaften 3 */
/* Autor:   Frank Eller           */
/* Sprache: C#                    */

using System;

public class cArtikel
{
  private double Price = 0;
  private string Name  = "";
  private double EUPrice = 0;
  private bool   AsEuro  = false;
  private const double cEuro = 1.95583;

  public string theName
  {
    get
    {
      return Name;
    }
    set
    {
      Name = value;
    }
  }

  public double thePrice
  {
    get
    {
      if (asEuro)
      {
        return EUPrice
      }
      else
      {
        return Price;
      }
    }
    set
    {
      if (asEuro)
      {
        EUPrice = value;
      }
      else
```

```
      {
        Price = value;
        EUPrice = value/cEuro;
      }
    }
  }

  public cArtikel(string Name, double Price, bool euro)
  {
    asEuro   = euro;
    theName  = Name;
    thePrice = Price;
  }

  public cArtikel(string Name, double Price)
  {
    aseuro   = false;
    thePrice = Price;
    theName  = Name;
  }
}
```

Sie finden ein Beispielprogramm auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_9\EIGENSCHAFTEN2.

Durch die einfache Änderung der Klasse haben wir alle Artikel plötzlich Euro-fähig gemacht. Wenn ein neuer Artikel angelegt oder ein Preis geändert wird, wird dieser automatisch auch in der Währung Euro festgelegt. Welche Art der Preisvergabe verwendet wird, kann der Anwender über die Eigenschaft `asEuro` festlegen. Sobald allerdings der Euro verwendet wird, wird der DM-Preis natürlich nicht mehr zugewiesen. Die Konstante mit dem Umrechnungswert von DM nach Euro verwenden wir natürlich für eben diese Umrechnung.

Wenn wir diese Klasse als Komponente compilieren würden (in eine DLL und auch in einen eigenen Namensraum), könnten später auch alle Visual Basic- oder C++-Programmierer damit arbeiten, ja, sie könnten die Klasse sogar erweitern bzw. umbauen. Das ist mitunter auch einer der größten Vorteile des .net-Frameworks.

## 9.2 Ereignisse von Klassen

Windows ist bekanntlich ein Betriebssystem, das auf Ereignisse reagieren kann. Wenn Sie als Anwender mit der Maus auf eine Schaltfläche klicken, wertet diese das als ein Ereignis. Ebenso ist es ein Ereignis, wenn Sie mit der Maus über einen Hyperlink fahren, was Sie

immer wieder im Internet beobachten können. Manchmal ändern sich Farben oder gleich ganze Bilder, wenn Sie die Maus darüberziehen.

Alle objektorientierten Systeme basieren auf einem solchen Verhalten. Eine komplett objektorientierte Sprache wie C# muss daher einen Mechanismus zur Verfügung stellen, solche Ereignisse auslösen zu können. Im Klartext: Eine Klasse in C# kann anderen Klassen mitteilen, dass gerade ein Ereignis ausgelöst wurde.

Hierfür wird allerdings ein besonderer Datentyp benötigt, nämlich ein so genannter *Delegate*. Wir haben die Delegates, ihre Deklaration und Verwendung bereits im letzten Kapitel durchgesprochen, so dass wir an dieser Stelle direkt in die Deklaration eines Ereignisses einsteigen können.

### 9.2.1 Das Ereignisobjekt

Um die Ereignisse innerhalb aller C#-Programme und -Klassen konform zu machen, hat man sich darauf geeinigt, dass ein Ereignis immer zwei bestimmte Parameter benötigt. Der erste Parameter ist dabei immer das Objekt, das das Ereignis auslöst. Der zweite Parameter ist ebenfalls ein Objekt, welches das Ereignis selbst beinhaltet. Solche Objekte sind stets abgeleitet von der Klasse `EventArgs`. Eine neue Klasse für ein Ereignis könnte also folgendermaßen aussehen:

```
/* Beispiel Ereignisse 1 */
/* Autor:   Frank Eller  */
/* Sprache: C#          */

using System;

class ChangedEventArgs : EventArgs
{
  string newValue;
  string msg;

  public ChangedEventArgs(string newValue,string msg)
  {
    this.newValue = newValue;
    this.msg      = msg;
  }

  public string NewValue
  {
    get
```

```
    {
      return (newValue);
    }
  }

  public string Msg
  {
    get
    {
      return (msg);
    }
  }
}
```

Damit haben wir bestimmt, welche Werte unser Ereignis übergeben bekommt. Wir können also, falls innerhalb einer Klasse ein Wert geändert wurde, erfahren, welches der neue Wert ist, und auch eine Nachricht übergeben, etwa in der Art von „Wert wurde geändert".

Die Klasse, die Sie von `EventArgs` ableiten, enthält nur die Daten, die an das Ereignis übergeben werden. Wenn Sie natürlich eine bestehende `EventArgs`-Klasse benutzen können, müssen Sie diese nicht für jedes Ereignis neu deklarieren. Es wird hier ohnehin nur die Art der Werte festgelegt, die an das Ereignis übermittelt werden.

Ein Beispiel wäre z.B. ein Mausklick. Dabei wird die linke Maustaste gedrückt und wieder gelöst. Diese beiden Ereignisse könnte man abfangen, unter Angabe beispielsweise der Mausposition. Das bedeutet, man benötigt für beide Ereignisse nur eine `EventArgs`-Klasse, da die übergebenen Werte ja vom gleichen Typ sind.

### 9.2.2 Die Ereignisbehandlungsroutine

Für die Ereignisbehandlung müssen wir nun einen Delegate deklarieren, der jeweils auf das korrekte Ereignis verweist. Wie bereits angesprochen benötigen wir für das Ereignis zwei Parameter, nämlich das Objekt, das unser Ereignis auslöst, und die Ereignisdaten, für deren Übergabe wir ja bereits eine Klasse erstellt haben. Das auslösende Objekt kann bekanntlich alles sein, also verwenden wir als Datentyp hierfür **object**.

```
public delegate void ChangedEventHandler(object Sender,
                                          ChangedEventArgs e);
```

Der nächste Schritt ist die Deklaration eines öffentlichen Felds mit dem Modifikator **event**, wobei der Typ unser Delegate für das Ereig-

nis sein muss. Dieses Feld stellt das Ereignis dar, das ausgelöst werden soll. Den Delegate verwenden wir, um die Ereignisbehandlungsroutine frei festlegen zu können, und natürlich, um festzulegen, welche Parameter an das Ereignis übergeben werden.

*Syntax*

```
public event <Delegate> : Bezeichner;
```

<Delegate> steht natürlich für den von uns zuvor deklarierten Delegate.

Wir haben nun ein Objekt für das Ereignis und wir haben mittels eines Delegate festgelegt, welche Parameter an die Ereignisbehandlungsroutine übergeben werden. Nun kommen wir zur Deklaration des eigentlichen Ereignisses, welches über das reservierte Wort **event** deklariert wird.

```
public event ChangedEventHandler OnChangedHandler;
```

Damit haben wir das eigentliche Ereignis festgelegt. Der Grund, warum wir auf diese etwas kompliziert anmutende Weise vorgehen müssen, ist darin zu suchen, dass wir nun kontrollieren können, ob für das Ereignis wirklich eine Methode deklariert und zugewiesen wurde. Wenn nicht, interessiert das Ereignis nämlich nicht und wir müssen auch keinen Code dafür bereitstellen. In einer einfachen Routine führen wir die Kontrolle durch:

```
protected void OnChanged(object sender, ChangedEventArgs e)
{
  if (OnChangedHandler != null)
    OnChangedHandler(sender,e);
}
```

Der Wert **null** steht bekanntlich für „nicht zugewiesen“, also ein leeres Objekt. Unser Delegate ist ebenfalls ein Objekt, und damit können wir auf einfache Art und Weise überprüfen, ob eine Ereignisbehandlungsroutine zugewiesen wurde (dann ist der Wert ungleich **null**) oder nicht.

Nun benötigen wir noch eine Routine, in der das Ereignis schließlich ausgelöst wird und die auch eine Instanz des Ereignisobjekts erstellt und initialisiert. Alles zusammen packen wir in eine Klasse, die nur dazu dient, das Ereignis aufzurufen.

```
public void Change(object Sender, string nval, string message)
{
  ChangedEventArgs e = new ChangedEventArgs(nval,message);
  OnChanged(sender,e);
}
```

Und hier die Deklaration der gesamten Klasse:

```
/* Beispiel Ereignisse 2 */
/* Autor:   Frank Eller  */
/* Sprache: C#          */

public class NotifyChange
{
  public event ChangedEventHandler OnChangedHandler;

  protected void OnChanged(object Sender,
                           ChangedEventArgs e)
  {
    if (OnChangedHandler != null)
      OnChangedHandler(Sender,e);
  }

  public void Change(object Sender, string nval,
                     string message)
  {
    ChangedEventArgs e = new ChangedEventArgs(nval,message);
    OnChanged(Sender,e);
  }
}
```

Natürlich fehlt nun noch die Klasse, die auf eine Änderung reagieren soll. In dieser werden wir auch eine Methode deklarieren, die im Falle einer Änderung ausgeführt wird. Diese Methode muss natürlich exakt die gleichen Parameter beinhalten wie auch unser Delegate, da es sich ja um eine solche Methode handelt. Außerdem müssen wir in unserer Klasse ein Feld vom Typ `NotifyChange` bereitstellen, da ansonsten eine Reaktion auf das Ereignis nicht möglich wäre.

Ich möchte an dieser Stelle zunächst den gesamten Quellcode der Klasse im Zusammenhang zeigen und danach auf die einzelnen Teile eingehen. Wir benutzen die bereits bekannte Klasse `cArtikel` und ändern sie ein wenig ab, damit das Ereignis ausgelöst wird.

```
/* Beispiel Ereignisse 3 */
/* Autor:   Frank Eller  */
/* Sprache: C#          */

public class cArtikel
{
  private double Price = 0;
  private string Name = "";
  private NotifyChange notifyChange;
```

```
  public string theName
  {
    get
    {
      return Name;
    }
    set
    {
      Name = value;
      notifyChange.Change(this,"Name geändert",value);
    }
  }

  public double thePrice
  {
    get
    {
      return Price;
    }
    set
    {
      Price = value;
      notifyChange.Change(this,"Preis geändert",
                          value.ToString());
    }
  }

  void hasChanged(object sender, ChangedEventArgs e)
  {
    //Wird beim Ereignisauftritt aufgerufen

    Console.WriteLine("{0}: Neuer Wert: {1}",
                      e.Msg,e.NewValue);
 }

  public cArtikel()
  {
    //Initialisierung des NotifyChange-Objekts

    this.notifyChange = new NotifyChange();
    notifyChange.OnChangedHandler +=
        new ChangedEventHandler(hasChanged);
  }
}
```

Die Klasse entspricht im Großen und Ganzen der ursprünglichen `cArtikel`-Klasse, lediglich bei den Zugriffsmethoden für die Eigen-

schaften und im Konstruktor der Klasse gibt es Änderungen. Neu hinzugekommen ist das Feld `notifyChange`, das wir benötigen, um unser Ereignis zu erzeugen. Weiterhin neu ist die Methode `hasChanged()`, die dann aufgerufen wird, wenn einer der Werte geändert wird. Das ist unsere Ereignismethode.

Im Konstruktor wird alles bereits vorbereitet. Es wird ein neues Objekt vom Typ `NotifyChange` erzeugt und dessen Eventhandler mittels des Delegate auf die Methode `hasChanged()` umgeleitet. Das ist eigentlich auch alles, was zu tun ist. Nun muss bei einer Änderung der Werte dem Objekt `notifyChange` nur noch mitgeteilt werden, dass eine Änderung stattgefunden hat. Prompt wird auch das Ereignis ausgelöst.

Die komplette Klasse incl. einer Hauptklasse zum Ausprobieren finden Sie auf der CD im Verzeichnis BEISPIELE\KAPITEL_9\EREIGNISSE.

Die meisten Ereignisse in Windows müssen Sie nicht selbst auf diese Art deklarieren. Das ist lediglich dann interessant, wenn Sie eigene, komplexere Komponenten erstellen, die Sie dann später anderen Programmierern zur Verfügung stellen. In anderen Fällen, wenn Sie eigene Programme entwickeln und auf die bereits vorhandenen Komponenten im .net-Framework zurückgreifen können, werden Sie eine solche Vorgehensweise nicht benötigen. Nichtsdestotrotz sollte man sie kennen, denn möglicherweise müssen Sie einmal eine fremde Komponente für Ihre eigenen Bedürfnisse anpassen. Dann ist es wichtig, die Zusammenhänge zu kennen.

Ereignisse und Eigenschaften sind nicht auf eigenständige Komponenten beschränkt. Sie können sie natürlich auch in Ihren Klassen verwenden. Vor allem die Verwendung von Eigenschaften aufgrund der einfacheren Zugriffs- und Änderungsmöglichkeiten bietet sich an.

## 9.3 Zusammenfassung

Ereignisse benötigen Sie zwar nur dann, wenn Sie eigene Komponenten zur Verwendung im .net-Framework programmieren, die Verwendung der Eigenschaften ist allerdings eine nützliche Sache, da mit einer kleinen Programmänderung an der richtigen Stelle eine recht umfangreiche Wirkung erzielt werden kann.

## 9.4 Kontrollfragen

Wie auch in den vorherigen Kapiteln wieder einige Fragen zum Thema.

1. Welchen Vorteil bieten Eigenschaften gegenüber Feldern?
2. Wie werden die Zugriffsmethoden der Eigenschaften genannt?
3. Welcher Unterschied besteht zwischen Eigenschaften und Feldern?
4. Wie kann man eine Eigenschaft realisieren, die nur einen Lesezugriff zulässt?
5. Welchen Datentyp hat der Wert `value`?
6. Was ist ein Ereignis?
7. Welche Parameter werden für ein Ereignis benötigt?
8. Welches reservierte Wort ist für die Festlegung des Ereignisses notwendig?

## 9.5 Übungen

### Übung 1

Erstellen Sie eine Klasse, die einen Notendurchschnitt berechnen kann. Es soll lediglich die Anzahl der geschriebenen Noten eingegeben werden können, damit aber sollen sowohl der Durchschnitt als auch die Gesamtanzahl der Schüler ermittelt werden können. Realisieren Sie alle Felder mit Eigenschaften, wobei die Eigenschaften Durchschnitt und Schüleranzahl nur zum Lesen bereitstehen sollen.

# 10 Überladen von Operatoren

Wir haben in Kapitel 6 alle Operatoren durchgesprochen, die C# anbietet. Bezogen auf einen Datentyp haben diese Operatoren eine eindeutige Funktion, die wir verwenden können. Wenn Sie jedoch eine eigene Klasse erstellt haben, z.B. eine numerische Klasse, möchten Sie möglicherweise ein anderes Verhalten dieser Operatoren erzeugen.

Ein gutes Beispiel in diesem Zusammenhang wäre der Operator ^, der in anderen Programmiersprachen teilweise eine Potenzierung bedeutet. In C# müssten Sie dafür die Methoden `Math.Pow` verwenden. Wenn Sie jedoch einen eigenen numerischen Datentyp erzeugt haben, könnten Sie diesen Operator so umbauen, dass er wie eine Potenzierung wirkt (anstelle der exklusiv-oder-Verknüpfung).

## 10.1 Arithmetische Operatoren

Kümmern wir uns zunächst um die arithmetischen Operatoren, die zum Überladen zur Verfügung stehen. Diese Operatoren sind +, -, *, /, %, &, |, ^, >> und <<. Bei den anderen arithmetischen Operatoren ist Überladen nicht möglich, die Gründe dafür werden wir weiter hinten im Kapitel noch genauer erläutern. Kommen wir jedoch zunächst zur Syntax.

***Operatormethoden***

Für jeden Operator, der eine neue Funktion erhalten soll, muss eine Methode erstellt werden. Die Werte, mit denen gerechnet wird, werden als Übergabeparameter deklariert, der zurückgelieferte Wert sollte vom Datentyp her auch dem Typ entsprechen, für den die Methode geschrieben wird.

***operator***

Um C# mitzuteilen, dass es sich hierbei um einen Operator handelt, der überladen werden soll, benutzen wir, wie wir es bereits gewohnt sind, einen Modifikator, nämlich **operator**. Weiterhin muss die Me-

thode als **static** deklariert werden, denn die Behandlung eines Operators obliegt nicht einer Instanz, sondern ist für die gesamte Klasse und alle von ihr abgeleiteten Objekte gleichermaßen gültig. Damit ergibt sich für das Überladen eines Operators folgende Syntax:

***Syntax***

```
public static <Typ> operator <Operator>(<Parameter>)
```

An einem Beispiel wird alles immer etwas besser deutlich als beim reinen Lesen, deshalb wollen wir nun einen neuen Datentyp deklarieren und bei diesem einige Operatoren programmieren. Der Operator ^, den wir bereits angesprochen haben, soll so umgeschrieben werden, dass es sich um einen Operator zur Potenzierung des in unserer Variable enthaltenen Werts handelt. Unser neuer Datentyp soll sich in etwa verhalten wie der Datentyp **int**.

```
/* Beispielklasse Operatoren überladen */
/* Autor:   Frank Eller                */
/* Sprache: C#                         */

using System;

public struct myInt
{
  int value;

  public myInt(int value)
  {
    this.value = value;
  }

  public override string ToString()
  {
    return (value.ToString());
  }

  public static myInt operator +(myInt v1, myInt v2)
  {
    return (new myInt(v1.value+v2.value));
  }

  public static myInt operator -(myInt v1, myInt v2)
  {
    return (new myInt(v1.value-v2.value));
  }
```

```
  public static myInt operator ++(myInt v1)
  {
    return (new myInt(v1.value+1));
  }

  public static myInt operator --(myInt v1)
  {
    return (new myInt(v1.value-1));
  }

  public static myInt operator ^(myInt v1, myInt v2)
  {
    double i = Math.Pow((double)(v1.value),
                        (double)(v2.value));
    return (new myInt((int)(i)));
  }
}
```

Wie Sie sehen, ist prinzipiell nicht besonders viel dabei. Es werden an sich nur der Operator deklariert, die Parameter angegeben und dann der entsprechende Wert, der sich bei der jeweiligen Berechnung ergibt, zurückgeliefert. Wichtig ist in diesem Fall, dass mittels **new** eine neue Instanz unserer Klasse erzeugt werden muss, wollen wir sie zurückliefern.

***nicht überladbare Operatoren***

In diesem Zusammenhang soll nicht unerwähnt bleiben, dass einige Operatoren keineswegs überladen werden können. Unter anderem handelt es sich hierbei um die zusammengesetzten arithmetischen Operatoren, denn diese sind auch in den anderen Datentypen nicht extra deklariert. Stattdessen werden sie vom Compiler so behandelt, als seien Zuweisung und Rechenoperation getrennt. C# macht das automatisch, Sie müssen sich also nicht darum kümmern.

Sie können unser kleines Beispiel auch testen. Wenn Sie z.B. den umprogrammierten Operator ^ verwenden, vielleicht sogar noch als zusammengesetzten Operator, werden Sie feststellen, dass er tatsächlich nun als Potenzierungsoperator arbeitet.

```
/* Beispiel Operatoren überladen 2 */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

public class TestClass
{
  public static void Main()
  {
    myInt x = new myInt(12);
```

```
    myInt y = new myInt(2);
    myInt z = new myInt(0);

    z = x^y;
    Console.WriteLine("Wert: {0}",z.ToString());
  }
}
```

Wenn Sie das Programm so eingeben (natürlich mit unserem **struct**), erhalten Sie als Ergebnis den korrekten Wert 144. Die Castings gerade in der Methode für den Operator ^ sind notwendig, weil wir in diesem Fall eine Methode der Klasse `Math` benutzen, die mit **double**-Werten arbeitet, unser Datentyp aber nur mit **int**-Werten umgehen kann.

Ein Beispielprogramm finden Sie auch auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_10\OPERATOREN1.

## 10.2 Konvertierungsoperatoren

Wir haben jetzt zwar einen Datentyp, aber wenn wir versuchen, diesem eine Zahl zuzuweisen (was eine implizite Konvertierung bedeuten würde), oder auch ein Casting versuchen, werden wir feststellen, dass es nicht funktioniert. Auch für die Konvertierung gilt, dass wir hierfür Methoden bereitstellen müssen. Für die implizite Konvertierung wird dabei der Modifikator **implicit** benutzt, für das Casting der Modifikator **explicit**. Wenn wir unseren **struct** erweitern, sieht es folgendermaßen aus:

```
/* Beispiel Operatoren überladen 3 */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

using System;

public struct myInt
{
  int value;

  public myInt(int value)
  {
    this.value = value;
  }

  public override string ToString()
  {
```

```
    return (value.ToString());
  }
  public static myInt operator +(myInt v1, myInt v2)
  {
    return (new myInt(v1.value+v2.value));
  }

  public static myInt operator -(myInt v1, myInt v2)
  {
    return (new myInt(v1.value-v2.value));
  }

  public static myInt operator ++(myInt v1)
  {
    return (new myInt(v1.value+1));
  }

  public static myInt operator --(myInt v1)
  {
    return (new myInt(v1.value-1));
  }

  public static myInt operator ^(myInt v1, myInt v2)
  {
    double i = Math.Pow((double)(v1.value),
                        (double)(v2.value));
    return (new myInt((int)(i)));
   }

  public static implicit operator int(myInt v1)
  {
    //implizit: von myInt nach int konvertieren
    return (v1.value);
  }

  public static explicit operator myInt(int v1)
  {
    //explizit: von int nach myInt (Casting)
    return (new myInt(v1));
  }
}
```

Mit der impliziten Konvertierung konvertieren wir von unserem Datentyp in einen anderen, mit der expliziten Konvertierung von einem anderen Datentyp in den unseren, allerdings mittels Casting. Anders ist das nicht machbar, da eine implizite Konvertierung von

int nach myInt im Datentyp int programmiert werden müsste. Dieser ist jedoch versiegelt, es ist also nicht möglich.

Ein Beispielprogramm finden Sie auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_10\OPERATOREN2.

Bei der Deklaration eines impliziten Operators müssen Sie immer darauf achten, dass zwei Dinge auf keinen Fall passieren dürfen: Es darf keine Exception ausgelöst werden, also kein Fehler auftauchen, und es dürfen keinerlei Daten verloren gehen. Sollte eines der beiden der Fall sein, ist es also z.B. möglich, dass Daten verloren gehen, benutzen Sie den Modifikator explicit und erzwingen Sie so ein Casting.

## 10.3 Vergleichsoperatoren

Um diese Überladung zu bewerkstelligen, müssen wir uns zunächst einige Gedanken machen.

Vergleichsoperatoren dienen dazu, festzustellen, ob zwei Werte identisch sind. Wenn wir diese Operatoren überladen wollen, müssen wir zunächst eine andere Funktion finden, die diesen Vergleich für uns durchführt, denn der Operator selbst kann ja nicht benutzt werden – es würde nämlich in diesem Fall die gleiche Methode aufgerufen, in der wir uns gerade befinden, was zu einer endlosen rekursiven Schleife führen würde.

***Equals()***

Glücklicherweise bietet C# mit der Methode Equals() eine Möglichkeit, die Werte zweier gleicher Objekte zu kontrollieren. Equals liefert einen booleschen Wert zurück, der dann true ist, wenn beide übergebenen Objekte (bzw. im Falle von Wertetypen die enthaltenen Werte) gleich sind. Für unser Beispiel gilt also, dass wir Equals() verwenden müssen, um die Vergleichsoperatoren zu überladen.

Ebenso können wir zum Überladen der Operatoren >, <, >= und <= nicht genau diese Operatoren benutzen, denn das wiederum würde darin resultieren, dass es zu einer rekursiven Endlosschleife kommt. Im folgenden Beispiel erweitern wir unsere bereits bestehende Klasse und fügen zwei Vergleichsoperatoren hinzu.

```
/* Beispiel Operatoren überladen 4 */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

using System;

public struct myInt
{
```

```
int value;

public myInt(int value)
{
  this.value = value;
}

public override string ToString()
{
  return (value.ToString());
}

public static myInt operator +(myInt v1, myInt v2)
{
  return (new myInt(v1.value+v2.value));
}

public static myInt operator -(myInt v1, myInt v2)
{
  return (new myInt(v1.value-v2.value));
}

public static myInt operator ++(myInt v1)
{
  return (new myInt(v1.value+1));
}

public static myInt operator --(myInt v1)
{
  return (new myInt(v1.value-1));
}

public static myInt operator ^(myInt v1, myInt v2)
{
  double i = Math.Pow((double)(v1.value),
                      (double)(v2.value));
  return (new myInt((int)(i)));
}

public static implicit operator int(myInt v1)
{
  //implizit: von myInt nach int konvertieren
  return (v1.value);
}
```

```
  public static explicit operator myInt(int v1)
  {
    //explizit: von int nach myInt (Casting)
    return (new myInt(v1));
  }

  public static bool operator ==(myInt v1, myInt v2)
  {
    return (v1.value.Equals(v2.value));
  }

  public static bool operator !=(myInt v1, myInt v2)
  {
    return (!(v1.value.Equals(v2.value)));
  }
}
```

Damit hätten wir für unsere neue numerische Klasse bereits einige Operatoren deklariert.

Sinn und Zweck des Überladens ist in diesem Beispiel natürlich nicht besonders gut ersichtlich. In der Regel macht es auch kaum Sinn, arithmetische Operatoren zu überladen, es sein denn, man hätte wirklich eine Klasse deklariert, bei der ihre Verwendung Sinn macht und ganz bestimmten Gesetzmäßigkeiten gehorchen muss. Wesentlich öfter ist es sinnvoll, die Vergleichsoperatoren zu überladen, um Vergleiche zwischen Objekten eigener Klassen zu vereinfachen.

Als Beispiel soll eine Klasse dienen, in der eine Adresse gespeichert ist. Wenn wir als Menschen einen Vergleich machen, bei dem wir feststellen wollen, ob es sich um die gleiche Adresse bzw. den gleichen Namen handelt, würden wir nicht auf Groß- oder Kleinschreibung achten. Der Computer aber tut das sehr wohl, d.h. es könnte passieren, dass eine Adresse zweimal abgespeichert wird, obwohl es sich um den gleichen Namen handelt – nur, weil vielleicht ein Buchstabe kleingeschrieben ist.

Wir können dies umgehen, indem wir unsere eigene kleine Routine für den Vergleich schreiben bzw. einen Operator zur Verfügung stellen, der den Vergleich korrekt durchführt.

```
/* Beispiel Operatoren überladen 5 */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

public class cAdress
{
  public string Name;
  public string Vorname;
  public string Strasse;
  public string PLZ;
  public string Ort;

  public cAdress()
  {
    //Standard-Konstruktor
  }

  public cAdress(string n, string v, string s,
                 string p, string o)
  {
    this.Name = n;
    this.Strasse = s;
    this.PLZ     = p;
    this.Ort     = o;
  }
  public static bool operator ==(cAdress a, cAdress b)
  {
    bool isEqual = true;

    //Kontrolle
    isEqual = isEqual &&
       a.Name.ToUpper().Equals(b.Name.ToUpper()));
    isEqual = isEqual &&
       a.Vorname.ToUpper().Equals(b.Vorname.ToUpper)));
    isEqual = isEqual &&
       a.Strasse.ToUpper().Equals(b.Strasse.ToUpper()));
    isEqual = isEqual && (a.PLZ.Equals(b.PLZ));

    //Ort ist uninteressant, da PLZ bereits überprüft

    return (isEqual);
  }

  public static bool operator !=(cAdress a, cAdress b)
  {
    bool isEqual = true;
```

```
    //Kontrolle
    isEqual = isEqual &&
       a.Name.ToUpper().Equals(b.Name.ToUpper()));
    isEqual = isEqual &&
       a.Vorname.ToUpper().Equals(b.Vorname.ToUpper()));
    isEqual = isEqual &&
       a.Strasse.ToUpper().Equals(b.Strasse.ToUpper()));
    isEqual = isEqual && (a.PLZ.Equals(b.PLZ));

    //Ort ist uninteressant, da PLZ bereits überprüft

    return (!isEqual);
  }
}
```

Die Operatoren für den Vergleich sind nun überladen und verhalten sich entsprechend der neuen Vorgaben innerhalb unserer eigenen Methoden. Damit können wir ausschließen, dass zwei Adressen nur wegen eines irrtümlich groß- oder kleingeschriebenen Buchstaben als unterschiedlich angesehen werden.

***Eigene Methode Equals()***

Eine zweite Möglichkeit, die es ebenfalls vereinfacht, ist das direkte Überschreiben der `Equals()`-Methode, d.h. wir stellen für unsere eigene Klasse eine neue `Equals()`-Methode zur Verfügung, die sich so verhält, wie wir es wünschen. Da es sich um eine virtuelle Methode handelt, ist die Implementierung trivial.

```
/* Beispiel Operatoren überladen 6 */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

public class cAdress
{
  public string Name;
  public string Vorname;
  public string Strasse;
  public string PLZ;
  public string Ort;

  public cAdress()
  {
    //Standard-Konstruktor
  }

  public cAdress(string n, string v, string s,
                 string p, string o)
  {
```

```
    this.Name = n;
    this.Strasse = s;
    this.PLZ     = p;
    this.Ort     = o;
  }
  public new bool Equals(cAdress a, cAdress b)
  {
    bool isEqual = true;

    //Kontrolle
    isEqual = isEqual &&
       a.Name.ToUpper().Equals(b.Name.ToUpper()));
    isEqual = isEqual &&
       a.Vorname.ToUpper().Equals(b.Vorname.ToUpper()));
    isEqual = isEqual &&
       a.Strasse.ToUpper().Equals(b.Strasse.ToUpper()));
    isEqual = isEqual && (a.PLZ.Equals(b.PLZ));

    //Ort ist uninteressant, da PLZ bereits überprüft

    return (isEqual);
  }

  public static bool operator ==(cAdress a, cAdress b)
  {
    return (a.Equals(b));
  }

  public static bool operator !=(cAdress a, cAdress b)
  {
    return (!(a.Equals(b)));
  }
}
```

**Paarweise Operatoren**

Die Operatoren == und != treten, wie Sie sehen können, immer paarweise auf. Und, das ist sehr wichtig, sie müssen auch immer paarweise überschrieben werden. Sie können also nicht eine neue Funktionalität für == zur Verfügung stellen und die alte für != beibehalten. Natürlich hat das einen guten Grund.

Die Operatoren arbeiten nicht nur mit Werten, sie lassen sich auch auf Objekte anwenden. Und in diesem Fall gibt es nicht nur die Unterscheidung, ob ein Objekt gleich oder ungleich dem anderen ist – es gibt weiterhin die Möglichkeit, dass ein Objekt derzeit ungültig ist (sein Wert entspricht dann **null**). Im Allgemeinen gilt für alle Werte:

`a == b entspricht !(a!=b).`

Für den Fall, dass ein Objekt **null** ist, gilt diese Zuordnung nicht mehr, wodurch sich ein komplett fehlerhaftes Verhalten der Operatoren ergeben könnte.

Der zweite, wesentlich trivialere Grund ist, dass ein Anwender, der mit einem Ihrer Objekte arbeitet, durchaus erwarten kann, dass == äquivalent zu != arbeitet. Sie müssen diese Operatoren also immer paarweise überladen.

Die Operatoren == und != müssen, wenn sie überladen werden, immer gemeinsam überladen werden. Der Versuch, nur einen der beiden Operatoren zu überladen, wird vom Compiler mit einer Fehlermeldung beantwortet.

## 10.4 Zusammenfassung

Das Überladen von Operatoren, sowohl von Rechenoperatoren als auch von Konvertierungs- oder Vergleichsoperatoren, kann durchaus Sinn machen. Es ist auch, wie man an den Beispielen sehen kann, relativ einfach zu bewerkstelligen. Allerdings sollten Sie bei der Verwendung dieser Möglichkeiten darauf achten, dass es für den Anwender klar ist, welche Funktion ein Operator ausführt.

## 10.5 Kontrollfragen

Wie üblich auch in diesem Kapitel einige Fragen, die der Vertiefung des Stoffes dienen sollen.

1. Warum können zusammengesetzte Operatoren nicht überladen werden?
2. Warum macht es kaum Sinn, arithmetische Operatoren zu überladen?
3. Welche der Vergleichsoperatoren müssen immer paarweise überladen werden?
4. Mit welcher Methode können die Werte zweier Objekte verglichen werden?
5. Welches Schlüsselwort ist dafür zuständig, einen Konvertierungsoperator für Casting zu deklarieren?

# 11 Fehlerbehandlung

Kein Programm ist ohne Fehler. Diese Weisheit gilt schon seit den Anfängen der Computertechnik und ist gerade heute, wo die Programme immer umfangreicher und leistungsfähiger werden, erst recht wahr. Trotzdem sollte man sein Programm immer so schreiben, dass die größte Anzahl möglicher Fehler abgefangen wird.

So soll der Anwender beispielsweise durchaus einmal eine Falscheingabe machen dürfen, ohne dass sich gleich das ganze Programm verabschiedet. Oder die Rechenfunktionen eines Programms sollen so ausgelegt sein, dass sie ungültige Werte erkennen und die Berechnung abbrechen – möglichst nicht ohne dem Anwender des Programms mitzuteilen, dass sich gerade ein Fehler ereignet hat.

***Exceptions***

C# nutzt für auftretende Programmfehler so genannte Exceptions (Ausnahmen). Wenn ein Fehler auftritt, wird ein entsprechendes Exception-Objekt erzeugt und der Fehler in einem Fenster angezeigt. Da eine Exception dies automatisch erledigt, ist es durchaus sinnvoll, diesen Mechanismus zur Information des Anwenders zu nutzen. Da Exceptions wie alles andere auch nur Klassen sind, können Sie Ihre eigene Exception davon ableiten und aufrufen, falls der entsprechende Fehler auftritt. Kümmern wir uns jedoch zunächst um das Abfangen einer Exception innerhalb des Programms.

## 11.1 Exceptions abfangen

Der wohl am einfachsten zu reproduzierende Fehler ist vermutlich die Division durch Null. Da dies ein mathematisch nicht erlaubter Vorgang ist, reagiert das .net-Framework darauf mit einer Fehlermeldung, die in diesem Fall aus einer `DivideByZeroException` besteht. Wie bereits gesagt, wird der Fehler angezeigt, die Methode, in der sich der Fehler ereignete, wird automatisch abgebrochen.

### 11.1.1 Die try-catch-Anweisung

Was aber, wenn Sie im Falle eines solchen Fehlers auch noch eine Log-Datei führen wollen? Immerhin wird die Methode nach Auftreten des Fehlers verlassen, der Fehler selbst als Objekt der Klasse Exception verschwindet ebenfalls, sobald er nicht mehr benötigt wird. Wie fast immer bietet C# auch hier eine Lösung in Form eines **try-catch**-Blocks. Eigentlich handelt es sich um zwei Blöcke, einmal den **try**-Block, in dem die Anweisungen ausgeführt werden, bei denen es zum Fehler kommen kann, und einmal den **catch**-Block, in den verzweigt wird, wenn ein Fehler aufgetreten ist. Innerhalb des **catch**-Blocks haben Sie dann die Möglichkeit, auf den Fehler zu reagieren, in eine Log-Datei zu schreiben oder andere Dinge zu tun, die Ihnen sinnvoll erscheinen. Die einfachste Form eines solchen **try-catch**-Blocks sieht folgendermaßen aus:

```
/* Beispiel try-catch-Block */
/* Autor:   Frank Eller     */
/* Sprache: C#              */

class TestClass
{
  int doDivide(int a, int b)
  {
    try
    {
      return (a/b);
    }

    catch(Exception e)
    {
      Console.WriteLine("Ausnahmefehler: {0}",e.Message);
    }
  }

  public static void Main()
  {
    int a = Console.ReadLine();
    int b = Console.ReadLine();
    int x = doDivide(a,b);
  }
}
```

Bei der Ausführung des Programms wird möglicherweise durch Null dividiert, damit wird auch die Exception `DivideByZeroException` ausgelöst werden. Das Programm springt dann sofort in den **catch**-Block,

durch dessen Übergabeparameter die zu behandelnde Exception genauer spezifiziert wird. Im obigen Fall ist die Basisklasse aller Exceptions, `Exception`, selbst angegeben, womit der `catch`-Block für alle ausgelösten Exceptions angesprungen würde.

*catch-Block*

Im `catch`-Block geben wir eine Nachricht für den Benutzer unseres Programms aus, in diesem Fall, dass es einen Ausnahmefehler gegeben hat und welcher Art er ist. Die ausgelöste Exception bringt diese Nachricht in der Eigenschaft `Message` mit, Sie könnten aber, falls Sie genau wissen, welche Exception ausgelöst wird, auch eine eigene Nachricht ausgeben. Zu den einzelnen Eigenschaften einer Exception kommen wir noch später in diesem Kapitel.

Wenn eine Exception innerhalb eines `try`-Blocks auftritt, springt das Programm in den `catch`-Block, führt die dort enthaltenen Anweisungen aus und verlässt die Methode, in der der Fehler aufgetreten ist. Die danach folgenden Anweisungen werden nicht mehr ausgeführt. Andererseits wird aber bei einer fehlerlosen Abhandlung des `try`-Blocks der `catch`-Block nicht abgehandelt und das Programm wird dahinter fortgesetzt.

### 11.1.2 Exceptions kontrolliert abfangen

Die `try-catch`-Anweisung im ersten Beispiel hat jeden auftretenden Fehler abgefangen, ganz gleich welcher Art er war, da wir für unseren `catch`-Block keine spezielle Exception angegeben haben. Es gibt jedoch in der Programmierung immer wieder Fälle, bei denen man für unterschiedliche Fehler eine unterschiedliche Behandlung wünscht. Natürlich könnte man alle Fehler innerhalb des gleichen `catch`-Blocks abhandeln, das ist aber aus Gründen der Übersichtlichkeit nicht der beste Weg. Stattdessen können Sie mehrere `catch`-Blöcke deklarieren und die Art der Exception, die in diesem Block behandelt wird, präzisieren.

```
/* Beispiel Exceptions präzisieren */
/* Autor:   Frank Eller            */
/* Sprache: C#                     */

class TestClass
{
  int doDivide(int a, int b)
  {
    try
    {
      return (a/b);
    }
```

```
    catch(DivideByZeroException e)
    {
      //Für Division durch Null ...
      Console.WriteLine("Durch Null darf man nicht ...");
    }

    catch(Exception e)
    {
      //Für alle anderen Exceptions ...
      Console.WriteLine("Ausnahmefehler: {0}",e.Message);
    }
  }

  public static void Main()
  {
    int a = Console.ReadLine();
    int b = Console.ReadLine();
    int x = doDivide(a,b);
  }
}
```

Das gleiche Programm, allerdings diesmal mit einer präzisierten catch-Anweisung. Falls bei der Ausführung nun eine `DivideByZeroException` auftritt, wird das Programm in den entsprechenden **catch**-Block verzweigen, für alle anderen Exceptions in den allgemeinen **catch**-Block. Es wird jedoch nicht beide **catch**-Blöcke abarbeiten, d.h. entweder den einen oder den anderen, aber nach Beendigung des ersten passenden **catch**-Blocks wird die Methode in jedem Fall verlassen.

Wenn Sie präzisierte **catch**-Blöcke verwenden, müssen Sie diese vor einem eventuellen allgemeingültigen **catch**-Block programmieren. Falls eine Exception auftritt, sucht das Programm sich den ersten möglichen **catch**-Block, d.h. wenn der allgemeingültige zuerst angegeben wird, wird er auch immer angesprungen.

### 11.1.3 Der try-finally-Block

Manchmal ist es notwendig, trotz einer auftretenden Exception noch gewisse Vorgänge innerhalb der Methode durchzuführen. Das kann möglich sein, wenn z.B. noch eine Datei geöffnet ist. Wir wissen, dass bei der Behandlung eines Fehlers die Methode sofort verlassen wird, d.h. wir müssten diese Vorgänge sowohl im **catch**-Block programmieren als auch außerhalb desselben, da er ja bei einer fehlerfreien Ausführung nicht angesprungen wird. Das ist unbefriedigend, da es mehr Programmierarbeit bedeutet und die gleichen Anweisungen

doppelt programmiert werden müssten. Für solche Fälle gibt es die **finally**-Anweisung, die in jedem Fall ausgeführt wird, ganz gleich, ob ein Fehler auftritt oder nicht.

```
/* Beispiel try-finally-Block */
/* Autor:   Frank Eller       */
/* Sprache: C#               */

class TestClass
{
  int doDivide(int a, int b)
  {
    try
    {
      return (a/b);
    }

    finally
    {
      //wird immer ausgeführt ...
      Console.WriteLine("Der finally-Block ...");
    }
  }

  public static void Main()
  {
    int a = Console.ReadLine();
    int b = Console.ReadLine();
    int x = doDivide(a,b);
  }
}
```

Der **finally**-Block im obigen Programm wird immer ausgeführt, unabhängig davon, ob eine Exception ausgelöst wird oder nicht. Die Anweisungen, die in jedem Fall ausgeführt werden müssen, müssen auf diese Weise nur einmal programmiert werden.

### 11.1.4 Die Verbindung von catch und finally

Wenn Sie mit Ausnahmefehlern, also mit Exceptions, hantieren, wird es sehr oft vorkommen, dass Sie sowohl einen oder mehrere **catch**-Blöcke ausführen lassen wollen und danach eine gewisse Bereinigung mittels eines **finally**-Blocks vornehmen. Zu diesem Zweck schreiben Sie die beiden Blöcke einfach hintereinander.

```
/* Beispiel try-catch-finally */
/* Autor:   Frank Eller       */
/* Sprache: C#               */

class TestClass
{
  int doDivide(int a, int b)
  {
    try
    {
      return (a/b);
    }

    catch(DivideByZeroException e)
    {
      //Für Division durch Null ...
      Console.WriteLine("Durch Null darf man nicht ...");
    }

    catch(Exception e)
    {
      //Für alle anderen Exceptions ...
      Console.WriteLine("Ausnahmefehler: {0}",e.Message);
    }

    finally
    {
      //wird immer ausgeführt ...
      Console.WriteLine("Der finally-Block ...");
    }
  }

  public static void Main()
  {
    int a = Console.ReadLine();
    int b = Console.ReadLine();
    int x = doDivide(a,b);
  }
}
```

Wenn Sie hier als zweite Zahl eine 0 eingeben, wird die Ausgabe des Programms lauten:

**Durch Null darf man nicht ...**

**Der finally-Block ...**

Zunächst wird also der **catch**-Block ausgeführt, daran angeschlossen der **finally**-Block. Dieses ist auch die übliche Vorgehensweise bei der Programmierung bzw. der Behandlung von Exceptions.

### 11.1.5 Exceptions weiterreichen

Sie müssen den **try-catch**-Block nicht innerhalb der Methode programmieren, in der er vorkommt. Sie können ebenso in der aufrufenden Methode die Fehlerbehandlung implementieren. Sehen Sie sich das folgende Beispiel an.

```
/* Beispiel Exceptions weiterreichen */
/* Autor:   Frank Eller             */
/* Sprache: C#                      */

public class TestClass
{
  int doDivide(int a, int b)
  {
    return (a/b);
  }

  public static void Main()
  {
    int a = Console.ReadLine();
    int b = Console.ReadLine();
    try
    {
      int x = doDivide(a,b);
    }
    catch(DivideByZeroException e)
    {
      Console.WriteLine("Catch-Block ... ");
    }
  }
}
```

Die Frage ist, ob der **catch**-Block hier bei einem Fehler auch ausgeführt wird. Um es vorweg zu sagen: Er wird ausgeführt. Wenn wir den Programmablauf durchgehen (immer vorausgesetzt, als zweite Zahl würde eine 0 eingegeben, damit es auch wirklich zu einem Fehler kommt), tritt in der Methode `doDivide()` eine Exception auf, die aber innerhalb dieser Methode nicht abgefangen wird. Die Methode wird verlassen, die Exception aber nicht abgehandelt.

Zurück in der Methode Main() haben wir einen catch-Block, der die aufgetretene Exception behandelt. Dieser wird aufgerufen und die Methode (in diesem Fall mit ihr das Programm) verlassen. Eine Exception wird also, falls kein try-catch-Block vorgefunden wird, weitergereicht, bis einer gefunden wird. Letzte Instanz ist eine globale Fehlerbehandlungsroutine, nämlich die, die eine auftretende Exception auch dann anzeigt, wenn keinerlei Feherbehandlung programmiert wurde. In der Regel wird das Programm dann auch beendet.

## 11.2 Eigene Exceptions erzeugen

Alle Exceptions im .net-Framework sind von der Klasse Exception abgeleitet. Wir können bereits selbst eigene Klassen von bereits bestehenden ableiten, also sollten wir auch in der Lage sein, eine eigene Exception zu erstellen und auszulösen. Das Auslösen wird uns in Kapitel 11.3 beschäftigen. An dieser Stelle werden wir uns erst darum kümmern, eine eigene Exception zu erstellen.

Da die Fehlerbehandlung nur für Klassen funktioniert, die von der Klasse Exception abgeleitet sind, müssen wir unsere eigene Fehlerklasse ebenfalls davon ableiten. Die Klasse Exception stellt dabei verschiedene Konstruktoren zur Verfügung, die Sie überladen sollten.

```
/* Beispiel Eigene Exceptions */
/* Autor:   Frank Eller       */
/* Sprache: C#               */

public class myException : Exception
{
  public myException : base()
  {
    //Standard-Konstruktor
  }

  public myException(string message) : base(message)
  {
    //Konstruktor unter Angabe einer Nachricht
  }

  public myException(string message,
                     Exception inner) : base(message,inner)
  {
    //Konstruktor unter Angabe einer Nachricht und der
    //vorhergehenden (inneren) Exception
  }
}
```

Der erste Konstruktor ist der Standard-Konstruktor, bei dem automatisch eine Nachricht vergeben wird. Der zweite Konstruktor ermöglicht zusätzlich die Angabe einer Nachricht, die dann durch die Fehlerbehandlungsroutine ausgegeben wird. Der dritte Konstruktor wiederum ermöglicht es, mehrere Exceptions nacheinander anzuzeigen, wobei die jeweils nachfolgende (innere) Exception mit angegeben wird.

## 11.3 Exceptions auslösen

Bisher haben wir lediglich von Exceptions gehört, die bei einem Fehler ausgelöst werden und die wir abfangen können, um entsprechenden Fehlerbehandlungscode zur Verfügung zu stellen. Wir wissen auch, dass im Falle einer Exception automatisch eine Ausgabe durch die globale Fehlerbehandlungsroutine erfolgt. Es ist immer sinnvoll, bereits vorhandene Mechanismen und Automatismen für eigene Zwecke zu nutzen, in diesem Fall also dafür zu sorgen, dass eine Exception ausgelöst wird, wodurch das .net-Framework die Ausgabe unseres Fehlers übernimmt.

*throw*

Das Auslösen einer Exception geschieht durch das reservierte Wort **throw**, gefolgt von der Instanz der Exception, die ausgelöst werden soll. Da es sich um ein Exception-Objekt handelt, das wir übergeben müssen, verwenden wir **new**:

```
/* Beispiel Exceptions auslösen */
/* Autor:   Frank Eller         */
/* Sprache: C#                  */

public class myException : Exception
{
  public myException : base()
  {
  }

  public myException(string message) : base(message)
  {
  }
}

public class TestClass
{
  public int doDivide(int a, int b)
  {
    if (b==0)
      throw new myException("Bereichsüberschreitung");
```

```
    else
      return (a/b);
  }

  public static void Main()
  {
    int a = Console.ReadLine();
    int b = Console.ReadLine();
    int x = doDivide(a,b);
  }
}
```

Im obigen Beispiel wird dann eine Exception ausgelöst, wenn der zweite eingegebene Wert gleich 0 ist. Es wird in diesem Fall keine Berechnung durchgeführt (was ebenfalls eine Exception ergeben würde, nämlich *DivideByZeroException*), sondern ein Fehler mit der Nachricht „Bereichsüberschreitung“ ausgegeben. Das .net-Framework macht dies automatisch.

## 11.4 Zusammenfassung

Exceptions sind eine recht intelligente Sache. Sie können mit Hilfe dieser Konstruktion sämtliche Fehler, die während eines Programmlaufs auftreten können, abfangen und darauf reagieren. Vor allem bei unerwünschten Aktionen des Anwenders (die leider dennoch oft auftreten) ist die Ausnahmebehandlung mit Hilfe von Exceptions eine gute Lösung.

Auch die Möglichkeit, eigene Exceptions erstellen und aufrufen zu können, ist eine sinnvolle Sache. So können Sie für eigene Fehler, die nicht bereits durch die Standard-Exceptions des .net-Frameworks abgedeckt sind, auch eigene Ausnahmen erstellen und aufrufen. Da die Anzeige dennoch automatisch und standardisiert erfolgt, geben Sie dadurch Ihren Programmen auch dann ein professionelles Aussehen, wenn es zu einem Fehler kommen sollte.

## 11.5 Kontrollfragen

Und wieder einige Fragen zum Thema.

1. Welche Anweisungen dienen zum Abfangen von Exceptions?
2. Von welcher Klasse sind alle Exceptions abgeleitet?
3. Was ist der Unterschied zwischen `try-catch` und `try-finally`?
4. Wie können Exceptions kontrolliert abgefangen werden?
5. Durch welches reservierte Wort können Exceptions ausgelöst werden?
6. Was versteht man unter dem Weiterreichen von Exceptions?

# 12 Lösungen

## 12.1 Antworten zu den Kontrollfragen

### 12.1.1 Antworten zu Kapitel 2

**1.** *Warum ist die Methode* `Main()` *so wichtig für ein Programm?*

Die Methode `Main` bezeichnet den Einsprungpunkt eines Programms. Wenn diese Methode nicht in einem Programm enthalten ist, kann dieses Programm nicht gestartet werden.

**2.** *Was bedeutet das Wort* `public`*?*

`public` bedeutet öffentlich. Auf Felder bzw. Methoden, die als `public` deklariert sind, kann von außerhalb der Klasse, in der sie enthalten sind, zugegriffen werden.

**3.** *Was bedeutet das Wort* `static`*?*

Übersetzt bedeutet `static` statisch. Methoden bzw. Variablen, die mit dem Modifikator `static` deklariert sind, sind Bestandteil der Klasse und somit unabhängig von einer Instanz.

**4.** *Welche Arten von Kommentaren gibt es?*

In C# gibt es mehrzeilige Kommentare, die zwischen den Zeichen /* und */ stehen müssen, und Kommentare bis zum Zeilenende, die mit einem doppelten Schrägstrich (//) beginnen. Die beiden Arten können durchaus verschachtelt werden.

**5.** *Was bedeutet das reservierte Wort* `void`*?*

Das reservierte Wort `void` wird bei Methoden benutzt, die keinen Wert zurückliefern. Es bedeutet schlicht eine leere Rückgabe.

6. *Wozu dient die Methode* `ReadLine()`*?*

   Die Methode `ReadLine()` liest einen Wert von der Tastatur ein. Der Wert wird im `string`-Format zurückgeliefert.

7. *Wie kann ich einen Wert oder eine Zeichenkette ausgeben?*

   Zur Ausgabe eines Wertes oder einer Zeichenkette dient in C# die Methode `WriteLine()`, die als statische Methode in der Klasse `Console` deklariert ist.

8. *Was bedeutet {0}?*

   Innerhalb einer Zeichenkette stehen diese Werte in geschweiften Klammern für einen Platzhalter. Der Wert, der den Platzhalter bei der Ausgabe ersetzt, wird nach der eigentlichen Zeichenkette angegeben. Die Methode `WriteLine()` fügt die angegebenen Werte in ihrer Reihenfolge an der Stelle der Platzhalter ein.

   Da die Zählung bei 0 beginnt, handelt es sich hier um den ersten Platzhalter, seine Position wird also durch den ersten übergebenen Wert ersetzt.

9. *Was ist eine lokale Variable?*

   Als lokale Variablen bezeichnet man Variablen, die innerhalb eines Codeblocks deklariert und damit auch nur innerhalb dieses Blocks gültig sind. Von außerhalb der Methode kann auf diese Variablen bzw. ihre Werte nicht zugegriffen werden.

10. *Wozu werden Escape-Sequenzen benötigt?*

    Escape-Sequenzen werden innerhalb von Zeichenketten normalerweise für die Formatierung benutzt. Sie werden auch benutzt, um Sonderzeichen oder Zeichen, die innerhalb der Programmiersprache C# eine besondere Bedeutung haben, auszugeben.

### 12.1.2 Antworten zu Kapitel 3

1. *Von welcher Basisklasse sind alle Klassen in C# abgeleitet?*

   Alle Klassen in C# sind von der Basisklasse `object` abgeleitet.

2. *Welche Bedeutung hat das Schlüsselwort new?*

   `new` bedeutet „erstelle eine neue Kopie von“. Das Schlüsselwort wird benutzt, um eine neue Instanz einer Klasse, also ein Objekt, zu erzeugen.

3. *Warum sollten Bezeichner für Variablen und Methoden immer eindeutige, sinnvolle Namen tragen?*

   Anhand eines Bezeichners sollte auch immer der Verwendungszweck der jeweiligen Variablen erkannt werden. Dazu dienen sinnvolle Bezeichner. Außerdem wird die spätere Wartung bzw. eine etwaige Erweiterung des Programms erleichtert.

4. *Welche Sichtbarkeit hat das Feld einer Klasse, wenn kein Modifikator bei der Deklaration benutzt wurde?*

   Innerhalb einer Klasse wird die Sichtbarkeitsstufe `private` benutzt, wenn kein Modifikator angegeben wird.

5. *Wozu dient der Datentyp* `void`*?*

   Das reservierte Wort `void` wird bei Methoden benutzt, die keinen Wert zurückliefern. Es bedeutet schlicht eine leere Rückgabe.

6. *Was ist der Unterschied zwischen Referenzparametern und Werteparametern?*

   Eigentlich sagt der Name bereits alles aus. Wenn Werteparameter benutzt werden, wird auch nur der Wert übergeben. Das bedeutet für eine etwaige ursprüngliche Variable, dass ihr Wert unverändert bleibt, obwohl er einer Methode übergeben wurde. Im Falle von Referenzparametern wird eine Referenz auf diese ursprüngliche Variable übergeben, d. h. es wird tatsächlich der Wert der Variable selbst verändert.

7. *Welche Werte kann eine Variable des Typs bool annehmen?*

   Boolesche Variablen können nur die Werte `true` oder `false` annehmen.

8. *Worauf muss beim Überladen einer Methode geachtet werden?*

   Der Compiler muss die Möglichkeit haben, die überladenen Methoden zu unterscheiden. Da der Name der Methoden dazu nicht herangezogen werden kann (denn er ist ja gleich) muss der Unterschied anhand der Parameter festgestellt werden. Überladene Methoden müssen also unterschiedliche Parameter oder Ergebnistypen ausweisen.

9. *Innerhalb welchen Gültigkeitsbereichs ist eine lokale Variable gültig?*

   Eine lokale Variable ist innerhalb des Codeblocks gültig, in dem sie deklariert ist.

**10.** *Wie kann eine globale Variable deklariert werden, ohne das Konzept der objektorientierten Programmierung zu verletzen?*

Variablen, die sich bei der späteren Programmierung wie globale Variablen verhalten sollen, können als statische Variablen einer Klasse deklariert werden. Man muss dann nur noch dafür sorgen, dass diese Klasse aus dem gesamten Programm heraus angesprochen werden kann.

**11.** *Wie kann ich innerhalb einer Methode auf ein Feld einer Klasse zugreifen, selbst wenn eine lokale Variable existiert, die den gleichen Bezeichner trägt wie das Feld, auf das ich zugreifen will?*

Für diese Art des Zugriffs gibt es das reservierte Wort `this`, das eine Referenz auf die aktuelle Instanz einer Klasse darstellt. Wenn also auf das Feld `x` der Instanz zugegriffen werden soll, obwohl auch eine lokale Variable `x` existiert, kann dies über `this.x` realisiert werden.

**12.** *Wie kann ich einen Namensraum verwenden?*

Es gibt zwei Möglichkeiten. Entweder wird der Namensraum bei der Verwendung von Klassen, die darin deklariert sind, immer mit angegeben (z.B. `System.Console.WriteLine()`) oder der gesamte Namensraum wird mittels `using` eingebunden.

**13.** *Mit welchem reservierten Wort wird ein Namensraum deklariert?*

Mit dem reservierten Wort `namespace`.

**14.** *Für welchen Datentyp ist int ein Alias?*

`int` ist ein Alias für den Datentyp `Int32`, der im Namensraum `System` deklariert ist.

**15.** *In welchem Namensraum sind die Standard-Datentypen von C# deklariert?*

Alle Standard-Datentypen sind im Namensraum `System` deklariert.

### 12.1.3 Antworten zu Kapitel 4

**1.** *Welcher Standard-Datentyp ist für die Verwaltung von 32-Bit-Ganzzahlen zuständig?*

Es handelt sich um den Datentyp `int`, deklariert als `System.Int32`.

**2.** *Was ist der Unterschied zwischen impliziter und expliziter Konvertierung?*

Bei impliziter Konvertierung kann weder ein Fehler noch eine Verfälschung des Zielwertes auftreten, da der Zieldatentyp eine größere Genauigkeit aufweist als der Quelldatentyp. Anders ausgedrückt: Im Zieldatentyp ist mehr Speicherplatz vorhanden als im Quelldatentyp.

Umgekehrt ist es bei der expliziten Konvertierung möglich, dass der Zielwert verfälscht wird, da der Zieldatentyp eine kleinere Genauigkeit aufweist als der Quelldatentyp (bzw. einen kleineren Zahlenbereich abdeckt).

**3.** *Wozu dient ein* `checked`*-Programmblock?*

`checked` wird bei der expliziten Konvertierung, beim Casting, benutzt, um einen Konvertierungsfehler aufzuspüren. Wenn nach dem Casting der konvertierte Wert anders ist als der ursprüngliche Wert, wird eine Exception ausgelöst.

**4.** *Wie wird die explizite Konvertierung auch genannt?*

Die Antwort wurde quasi schon gegeben. Es handelt sich dabei um das *Casting*.

**5.** *Worin besteht der Unterschied zwischen den Methoden* `Parse()` *und* `ToInt32()` *bezogen auf die Konvertierung eines Werts vom Typ* `string`*?*

`Parse()` ist eine Klassenmethode des Datentyps, in den konvertiert werden soll, `ToInt32()` eine Instanzmethode des Datentyps `string`. `Parse()` hat außerdem den Vorteil, dass die landesspezifischen Einstellungen berücksichtigt werden.

**6.** *Wie viele Bytes belegt ein Buchstabe innerhalb eines Strings?*

Exakt zwei Bytes (16 Bit), da C# mit dem Unicode-Zeichensatz arbeitet.

**7.** *Was wird verändert, wenn das Zeichen @ vor einem* `string` *verwendet wird?*

Die Escape-Sequenzen werden nun nicht mehr beachtet. Das bedeutet, ein Backslash wird nur als Backslash-Zeichen angesehen und nicht mehr als Beginn einer Escape-Sequenz. Das kann der Erleichterung bei der Eingabe von Pfaden dienen.

8. *Welche Escape-Sequenz dient dazu, einen Wagenrücklauf durchzuführen (eine Zeile weiter zu schalten)?*

   Die Sequenz `\n`.

9. *Was bewirkt die Methode* `Concat()` *des Datentyps string?*

   Sie fügt mehrere Strings zu einem einzigen String zusammen.

10. *Was bewirkt das Zeichen # bei der Formatierung eines String?*

    Das Zeichen # steht als Platzhalter für eine Leerstelle, führend oder nachfolgend.

11. *Wie können mehrere Zeichen innerhalb einer Formatierungssequenz exakt so ausgegeben werden, wie sie geschrieben sind?*

    Um eine Zeichenfolge exakt so auszugeben, wie sie im Programmtext angegeben ist (und um sie nicht möglicherweise fehlerhaft als Formatierungszeichen zu interpretieren), setzen Sie sie in einfache Anführungszeichen.

12. *Was bewirkt die Angabe des Buchstabens* `G` *im Platzhalter bei der Formatierung einer Zahl, wie z.B. in* `{0:G5}`*?*

    Bei Angabe dieses Zeichens wird der Wert in dem Format ausgegeben, das die kompaktere Darstellung ermöglicht. Verwendet werden entweder das Gleitkommaformat oder die wissenschaftliche Notation.

### 12.1.4 Antworten zu Kapitel 5

1. *Wozu dient die* `goto`*-Anweisung?*

   Mit Hilfe der `goto`-Anweisung kann ein absoluter Sprung zu einem vorher deklarierten Label programmiert werden. Anweisungen, die sich zwischen der Anweisung `goto` und dem Label befinden, werden einfach übersprungen. Allerdings kann man mit `goto` nicht aus dem Gültigkeitsbereich einer Methode herausspringen.

2. *Welchen Ergebnistyp muss eine Bedingung für eine Verzweigung liefern, wenn die* `if`*-Anweisung benutzt werden soll?*

   Der Ergebnistyp muss `bool` (`System.Boolean`) sein.

3. *Welcher Datentyp muss für eine* `switch`*-Anweisung verwendet werden?*

   Für `switch`-Anweisungen werden in der Regel ganzzahlige (ordinale) Datentypen verwendet, es ist allerdings in C# ebenso möglich, einen String zu verwenden. Das funktioniert in anderen Programmiersprachen nicht.

4. *Wann spricht man von einer nicht-abweisenden Schleife?*

   Nicht-abweisende Schleifen sind alle Schleifen, bei denen die Anweisungen innerhalb der Schleife mindestens einmal durchlaufen werden, bevor die Schleifenbedingung kontrolliert wird.

5. *Wie müsste eine Endlosschleife aussehen, wenn sie mit Hilfe der* `for`*-Anweisung programmiert wäre?*

   Unter der Voraussetzung, dass die Schleife innerhalb des Schleifenblocks nicht durch die Anweisung break abgebrochen wird, kann eine Endlosschleife z.B. folgendermaßen programmiert werden:

```
for(;;)
{
  //Anweisungen
}
```

   Das ist allerdings nicht die einzige Möglichkeit, derer gibt es viele. Sie sollten dies aber nicht in eigenen Programmen testen, es wäre möglich, dass Sie mit einer solchen Schleife ihr System lahm legen.

6. *Was bewirkt die Anweisung* `break`*?*

   `break` verlässt den aktuellen Programmblock. Dabei kann es sich um jeden beliebigen Programmblock handeln, `break` ist allgemeingültig.

7. *Was bewirkt die Anweisung* `continue`*?*

   Die Anweisung `continue` ist nur in Schleifenblöcken gültig und bewirkt, dass der nächste Schleifendurchlauf gestartet wird. Dabei werden, im Falle einer `for`-Schleife, die Laufvariable weitergeschaltet (es wird die Aktualisierungsanweisung im Kopf der `for`-Schleife ausgeführt) und die Abbruchbedingung für die Schleife kontrolliert.

8. *Ist die Laufvariable einer* `for`*-Schleife, wenn sie im Schleifenkopf deklariert wurde, auch für den Rest der Methode gültig?*

   Nein, nur für den Schleifenblock.

9. *Wohin kann innerhalb eines* `switch`*-Anweisungsblocks mittels der* `goto`*-Anweisung gesprungen werden?*

   Innerhalb der `switch`-Anweisung kann man mit `goto` entweder zu einem `case`-Statement oder zum `default`-Statement springen, falls dieses vorhanden ist.

10. *Wie kann man innerhalb eines* `switch`*-Blocks mehreren Bedingungen die gleiche Routine zuweisen?*

    Zu diesem Zweck werden die `case`-Statements für die verschiedenen Bedingungen einfach untereinander geschrieben. Wenn das Programm ausgeführt wird, „fällt" der Compiler durch die einzelnen Statements, bis er zu den Anweisungen kommt (so genanntes *Fallthrough*).

11. *Warum sollte die bedingte Zuweisung nicht für komplizierte Zuweisungen benutzt werden?*

    Weil die Gefahr besteht, dass man sich beim Lesen der Anweisung dann die Augen verknotet ... ☺.

### 12.1.5 Antworten zu Kapitel 6

1. *Welchem Rechenoperator kommt in C# eine besondere Bedeutung zu?*

   Es handelt sich um den Divisionsoperator, der sowohl mit ganzzahligen Werten als auch mit Gleitkommawerten arbeitet.

2. *In welcher Klasse sind viele mathematische Funktionen enthalten?*

   In der Klasse `Math`, die im Namensraum `System` deklariert ist.

3. *Welche statische Methode dient der Berechnung der Quadratwurzel?*

   Die Methode `Math.Sqrt()`.

4. *Warum muss man beim Rechnen mit den Winkelfunktionen von C# etwas aufpassen?*

   Alle Winkelfunktionen von C# arbeiten im Bogenmaß, es ist also notwendig, die Werte vor bzw. nach der Berechnung umzuwandeln.

5. *Vergleichsoperatoren liefern immer einen Wert zurück. Welchen Datentyp hat dieser Wert?*

   Dieser Wert ist immer vom Typ `bool`.

6. *Was ist der Unterschied zwischen den Operatoren && und &?*

   Der erste ist ein logischer Operator, der zur Verknüpfung zweier boolescher Werte oder zweier Bedingungen dient. Der zweite ist ein bitweiser Operator, der auf Zahlen angewendet werden kann und die Zahlenwerte Bit für Bit miteinander verknüpft, so dass sich als Ergebnis ein neuer Wert ergibt.

7. *Mit welchem Wert wird beim Verschieben einzelner Bits einer negativen Zahl aufgefüllt?*

   Im Falle von negativen Zahlen wird mit 1 aufgefüllt, wenn die Zahl positiv ist, mit 0.

8. *Wie kann man herausfinden, ob das vierte Bit einer Zahl gesetzt ist?*

   Jedes Bit einer Zahl entspricht einem bestimmten Wert, nämlich dem Wert 2 potenziert mit der Position des Bit. Gezählt wird dabei von 0 an, d.h. das erste Bit entspricht dem Wert 1 ($2^0$), das zweite Bit dem Wert 2 ($2^1$) und das vierte Bit dementsprechend dem Wert 8 ($2^3$).

### 12.1.6 Antworten zu Kapitel 7

1. *Bis zu welcher Größe ist ein* `struct` *effektiv?*

   Bis ungefähr 16 Byte ist ein struct effektiver als eine Klasse.

2. *Was ist der Unterschied zwischen einem* `struct` *und einer Klasse?*

   Ein struct ist ein Wertetyp, eine Klasse ein Referenztyp. Ansonsten besteht im Prinzip kein Unterschied.

3. *Mit welchem Wert beginnt standardmäßig eine Aufzählung?*

   Wie fast immer bei Computern wird auch hier standardmäßig mit dem Wert 0 begonnen. Durch die Angabe des ersten Wertes (z.B. dem Wert 1) kann dies aber geändert werden.

4. *Welche Arten von Aufzählungstypen gibt es?*

   Es gibt grob zwei Arten von Aufzählungstypen, nämlich die herkömmlichen Typen, bei denen nur immer ein Element der Aufzählung ausgewählt werden kann, und die so genannten Flag-Enums, bei denen mehrere Elemente ausgewählt werden können.

5. *Wie kann der Datentyp, der für eine Aufzählung verwendet wird, angegeben werden?*

   Der zu verwendende Datentyp wird einfach vor die Elementliste der Aufzählung geschrieben.

6. *Auf welche Art können den Elementen einer Aufzählung unterschiedliche Werte zugewiesen werden?*

   Jedem Element einer Aufzählung kann ein eigener Wert zugewiesen werden, indem das Gleichheitszeichen (der Zuweisungsoperator) benutzt wird. Es ist auch möglich, mehrere Elemente mit dem gleichen Wert zu belegen.

### 12.1.7 Antworten zu Kapitel 8

1. *Von welchen Klassen muss in jedem Fall abgeleitet werden?*

   Von abstrakten Klassen. In diesen sind nicht alle Methoden implementiert, sondern manche (oder sogar alle) sind nur deklariert. In der abgeleiteten Klasse müssen all diese abstrakten Methoden implementiert werden, ob sie benötigt werden oder nicht.

2. *Mit welchem Modifikator werden Methoden deklariert, die von einer abgeleiteten Klasse überschrieben werden können?*

   Mit dem Modifikator `virtual`.

3. *Wozu dient der Modifikator* `override`*?*

   override und virtual hängen zusammen. Während Methoden, die überschrieben werden können, mit virtual deklariert werden, muss beim Überschreiben dieser Methoden das reservierte Wort override verwendet werden.

4. *Was ist die Eigenschaft einer versiegelten Klasse?*

   Versiegelte Klassen haben die Eigenschaft, dass von ihnen keine weitere Klasse abgeleitet werden kann.

5. *Woran kann man eine versiegelte Klasse erkennen?*

   Das Schlüsselwort, an dem eine versiegelte Klasse erkannt wird, ist das reservierte Wort `sealed`.

6. *Was ist der Unterschied zwischen abstrakten Klassen und Interfaces, von denen ja in jedem Fall abgeleitet werden muss?*

   Zunächst ist es so, dass ein Interface keinerlei Funktionalität beinhaltet, eine abstrakte Klasse schon. Zumindest ist es ihr möglich.

Der gravierendste Unterschied besteht aber in der Möglichkeit der Mehrfachvererbung. Eine neue Klasse kann immer nur eine Vorgängerklasse haben, sie kann aber beliebig viele Interfaces implementieren.

**7.** *Kann ein Interface Funktionalität enthalten?*

Diese Antwort wurde eigentlich schon bei der vorigen Frage gegeben. Ein Interface enthält lediglich Deklarationen, keine Funktionalität.

**8.** *Wie können Methoden gleichen Namens in unterschiedlichen Interfaces dennoch verwendet werden?*

In diesem Fall muss man das Interface, auf das man sich bezieht, explizit angeben. Man qualifiziert einfach den Methodenbezeichner, gibt also den Namen des zu verwendenden Interface mit an.

**9.** *Wie kann auf die Methoden eines Interface zugegriffen werden, das in einer abgeleiteten Klasse implementiert wurde?*

Durch Casting. Da die Klasse das Interface beinhaltet, kann eine Variable, die vom Typ des Interface ist, durch Casting auch unser Objekt aufnehmen. Dieses Casting ist notwendig, da die Methoden im Interface deklariert sind, auch wenn die Funktionalität in der Klasse programmiert wurde.

**10.** *Was bedeutet das Wort* `delegate`*?*

Wörtlich übersetzt bedeutet es so viel wie Abgesandter. Im Prinzip handelt es sich bei einem Delegate um die Möglichkeit, ähnlich eines Zeigers auf verschiedene Methoden zu verzweigen.

**11.** *Wozu dienen Delegates?*

Delegates können wie bereits angemerkt dazu verwendet werden, auf verschiedene Methoden zu verzweigen, wenn diese die gleichen Parameter haben. Am häufigsten werden Delegates allerdings bei der Deklaration von Ereignissen verwendet.

**12.** *Was ist die Entsprechung eines Delegate in anderen Programmiersprachen?*

Auch dies wurde bereits beantwortet. Es handelt sich um das Äquivalent eines Zeigers.

### 12.1.8 Antworten zu Kapitel 9

**1.** *Welchen Vorteil bieten Eigenschaften gegenüber Feldern?*

Der größte Vorteil ist natürlich, dass die Funktionalität einer Klasse vollständig von der Deklaration getrennt ist. Damit können innerhalb des Programms die Zuweisungsanweisungen gleich bleiben, während die eigentliche Zuweisung, die ja über Methoden realisiert ist, geändert werden kann.

**2.** *Wie werden die Zugriffsmethoden der Eigenschaften genannt?*

Es handelt sich um den *Getter* und den *Setter*. Die Bezeichner der Methoden sind dementsprechend `get` und `set`.

**3.** *Welcher Unterschied besteht zwischen Eigenschaften und Feldern?*

Der Unterschied ist natürlich, dass für die Zuweisung bzw. das Auslesen von Werten bei Eigenschaften Methoden verwendet werden, bei Feldern nicht.

**4.** *Wie kann man eine Eigenschaft realisieren, die nur einen Lesezugriff zulässt?*

Die get-Methode ist immer für das Auslesen eines Wertes zuständig, die set-Methode für das Setzen des Wertes. Wenn der Wert der Eigenschaft also nur zum Lesen sein soll, können Sie die Methode set einfach weglassen. Innerhalb der Klasse kann dann natürlich immer noch auf das Feld, das die Eigenschaft repräsentiert, zugegriffen werden. Von außen aber kann man den Wert der Eigenschaft nicht mehr ändern.

**5.** *Welchen Datentyp hat der Wert* `value`*?*

Der Datentyp von `value` entspricht immer dem Datentyp der Eigenschaft, in der das reservierte Wort verwendet wird.

**6.** *Was ist ein Ereignis?*

Alles, was der Anwender tut, führt im Prinzip zu einem Ereignis, auf das man als Programmierer reagieren kann. Dazu gehört ein Tastendruck, das Bewegen der Maus, das Klicken auf einen bestimmten Bereich usw. Nahezu jede Aktion des Anwenders repräsentiert ein Ereignis, das vom Betriebssystem bzw. von einem Programm ausgewertet werden kann.

7. *Welche Parameter werden für ein Ereignis benötigt?*

   Man hat sich darauf geeinigt, dass bei Ereignissen immer zwei bestimmte Parameter, zwei Objekte, übergeben werden. Das erste Objekt ist die Klasse bzw. die Komponente, die das Ereignis auslöst. Das zweite Objekt ist das Ereignis selbst in Form eines Ereignisobjekts. Diese Ereignisobjekte sind abgeleitet von der Klasse `EventArgs`.

8. *Welches reservierte Wort ist für die Festlegung des Ereignisses notwendig?*

   Das reservierte Wort **event**.

### 12.1.9 Antworten zu Kapitel 10

1. *Warum können zusammengesetzte Operatoren nicht überladen werden?*

   Zusammengesetzte Operatoren müssen nicht überladen werden, weil sie eigentlich nicht existieren. Wenn ein zusammengesetzter Operator benutzt wird, behandelt C# diesen so, als handle es sich um zwei Anweisungen, nämlich die Rechenoperation mit anschließender Zuweisung. Daher funktionieren diese zusammengesetzten Operatoren auch, wenn ein Rechenoperator überladen wurde.

2. *Warum macht es kaum Sinn, arithmetische Operatoren zu überladen?*

   Arithmetische Operatoren sind für den Anwender eindeutig, die entsprechende (oder erwartete) Funktion kann aufgrund des Zusammenhangs mit der Mathematik sofort erkannt werden. Deshalb macht es nur selten Sinn, das Verhalten dieser Operatoren zu ändern.

3. *Welche der Vergleichsoperatoren müssen immer paarweise überladen werden?*

   Es handelt sich um die Operatoren == und !=.

4. *Mit welcher Methode können die Werte zweier Objekte verglichen werden?*

   Mit der Methode `Equals()`.

5. *Welches Schlüsselwort ist dafür zuständig, einen Konvertierungsoperator für Casting zu deklarieren?*

   Für Konvertierungsoperatoren gibt es zwei Schlüsselwörter, nämlich **implicit** und **explicit**. Da das Casting eine explizite Konvertierung bedeutet, handelt es sich auch um das gleich lautende Schlüsselwort **explicit**.

### 12.1.10 Antworten zu Kapitel 11

1. *Welche Anweisungen dienen zum Abfangen von Exceptions?*

   Zum Abfangen gibt es entweder den `try-catch`-Anweisungsblock oder den `try-finally`-Anweisungsblock.

2. *Von welcher Klasse sind alle Exceptions abgeleitet?*

   Von der Klasse `Exception`.

3. *Was ist der Unterschied zwischen* `try-catch` *und* `try-finally`*?*

   Bei der Verwendung von `try-catch` wird der `catch`-Block nur dann abgehandelt, wenn auch wirklich eine Exception auftritt. Wenn Sie `try-finally` verwenden, wird der `finally`-Block in jedem Fall abgehandelt, gleich ob eine Exception auftritt oder nicht.

4. *Wie können Exceptions kontrolliert abgefangen werden?*

   Sie können bei jedem `catch`-Block in Klammern die Exception angeben, für die er angesprungen werden soll. Damit haben Sie eine Kontrolle darüber, bei welcher Exception welcher `catch`-Block angesprungen wird.

5. *Durch welches reservierte Wort können Exceptions ausgelöst werden?*

   Durch das reservierte Wort `throw`.

6. *Was versteht man unter dem Weiterreichen von Exceptions?*

   Wenn eine Exception in einem Programmblock auftritt, der durch einen `try`-Block gesch_tzt ist, dann sucht C# nach einem `catch`- oder `finally`-Block, den er abhandeln kann. Findet er diesen nicht, wird die aktuelle Methode nat_rlich verlassen, es wird in der vorhergehenden Methode aber weiterhin nach einem `catch`-Block gesucht. Wird einer gefunden, dann springt C# dort hinein. Damit wurde die Exception weitergereicht.

## 12.2 Lösungen zu den Übungen

### 12.2.1 Lösungen zu Kapitel 3

#### Übung 1

*Deklarieren Sie eine Klasse, in der Sie einen String-Wert, einen Integer-Wert und einen Double-Wert speichern können.*

```
class Uebungen
{
  public string theString;
  public int    theInteger;
  public double theDouble;

  public Uebungen()
  {
    //Standard-Konstruktor
  }
}
```

Die Lösung dieser Übung dürfte für sich selbst sprechen.

### Übung 2

*Erstellen Sie für jedes der drei Felder einen Konstruktor, so dass das entsprechende Feld bereits bei der Instanziierung mit einem Wert belegt werden kann.*

```
class Uebungen
{
  public string theString;
  public int    theInteger;
  public double theDouble;

  public Uebungen()
  {
    //Standard-Konstruktor
  }

  public Uebungen(string theValue)
  {
     this.theString = theValue;
  }

  public Uebungen(int theValue)
  {
    this.theInteger = theValue;
  }

  public Uebungen(double theValue)
  {
    this.theDouble = theValue;
  }

}
```

Da die drei Konstruktoren sich zwangsläufig in der Art der übergebenen Parameter unterscheiden, brauchen wir nichts weiter zu tun, als sie einfach hinzuschreiben.

### Übung 3

*Erstellen Sie eine Methode, in der zwei Integer-Werte miteinander multipliziert werden. Es soll sich dabei um eine statische Methode handeln.*

Auch diese Übung ist nicht weiter schwer. Die statische Methode sieht aus wie folgt:

```
public static int DoMultiply(int a, int b)
{
  return (a*b);
}
```

Sie müssen sie lediglich in die Klassendeklaration hineinschreiben. Danach können Sie sie mittels `Uebungen.DoMultiply()` aufrufen.

### Übung 4

*Erstellen Sie drei Methoden um den Feldern Werte zuweisen zu können. Der Name der drei Methoden soll gleich sein.*

Im Prinzip handelt es sich bei diesen Methoden um die gleiche Funktionalität, die auch die Konstruktoren zur Verfügung stellen. Da wir diese aber nur einmal aufrufen können, nämlich bei der Erzeugung eines Objekts, müssen wir die Funktionen eben nochmals programmieren.

```
public void SetValue(string theValue)
{
   this.theString = theValue;
}

public void SetValue(int theValue)
{
  this.theInteger = theValue;
}

public void SetValue(double theValue)
{
  this.theDouble = theValue;
}
```

Da die Funktionen sich in den Parametern unterscheiden und der Compiler somit die richtige Methode finden kann. müssen wir nichts weiter tun. Bei den drei neuen Methoden handelt es sich um überladene Methoden.

### Übung 5

*Erstellen Sie eine Methode, mit der einem als Parameter übergebenen String der in der Klasse als Feld gespeicherte String hinzugefügt werden kann. Um zwei Strings aneinander zu fügen, können Sie den +-Operator benutzen, Sie können sie also ganz einfach addieren. Die Methode soll keinen Wert zurückliefern.*

Der String, der sich verändern soll, soll als Parameter übergeben werden und die Methode soll keinen Wert zurückliefern. Damit bleibt als einzige Möglichkeit nur noch ein Referenzparameter. Mit diesem ist die Methode aber schnell geschrieben.

```
public void AddString(ref string theValue)
{
  theValue = theValue + this.theString
}
```

Natürlich ist es ebenso möglich, einen zusammengesetzten Operator zu benutzen (für diejenigen, die diese Art Operatoren bereits kennen). Die Methode sieht dann so aus:

```
public void AddString(ref string theValue)
{
  theValue += this.theString
}
```

## 12.2.2 Lösungen zu Kapitel 4

### Übung 1

*Erstellen Sie eine neue Klasse mit zwei Feldern, die int-Werte aufnehmen können. Stellen Sie Methoden zur Verfügung, mit denen diese Werte ausgegeben und eingelesen werden können. Standardmäßig soll der Wert der Felder 0 sein.*

Das alles haben wir schon einmal gemacht, es ist also nicht weiter schwer:

```
class ZahlenKlasse
{
  private int a = 0;
  private int b = 0;

  public int GetA()
  {
    return (a);
  }

  public int GetB()
  {
    return (b);
  }

  public void SetA(int a)
  {
    this.a = a;
  }

  public void SetB(int b)
  {
    this.b = b;
  }
}
```

### Übung 2

*Schreiben Sie eine Methode, in der Sie die beiden Werte dividieren. Das Ergebnis soll aber als double-Wert zurückgeliefert werden.*

Die Methode, die wir hinzufügen, nennen wir `DoDivide()`. Es ist hier allerdings nur die Methode aufgeführt, nicht mehr die gesamte Klasse.

```
public double DoDivide()
{
  double x = a;
  return (x/b);
}
```

Indem wir einen der beiden Werte, mit denen wir rechnen, zu einem `double`-Wert machen, ändern wir automatisch auch den Ergebnistyp der Berechnung. Dieser entspricht immer dem Datentyp mit der höchsten Genauigkeit in der Rechnung, in diesem Fall also `double`.

Ein Casting wird für die Konvertierung nicht benötigt, da `double` im Vergleich zu `int` der genauere Datentyp ist, einen `int`-Wert also einfach aufnehmen kann.

Natürlich müssen Sie diese Methode in die Klassendeklaration hineinschreiben.

### Übung 3

*Schreiben Sie eine Methode, die das Gleiche tut, den Wert aber mit drei Nachkommastellen und als string zurückliefert. Die vorherige Methode soll weiterhin existieren und verfügbar sein.*

Natürlich existiert die vorige Methode weiterhin, wir löschen sie ja nicht. Wir werden also eine neue Methode schreiben, die diesmal einen `string`-Wert zurückliefert.

```
public string StrDivide()
{
  double x = DoDivide();
  return (string.Format("{0:F3}",x));
}
```

Diese Methode unterscheidet sich nicht von der vorherigen, was die Rechnung betrifft. Deshalb können wir die vorherige auch einfach aufrufen. Das Ergebnis formatieren wir dann wie gewünscht und liefern es an die aufrufende Methode zurück.

Die Formatierung ist auch nicht weiter schwer zu verstehen. Wir formatieren als Nachkommazahl mit drei Nachkommastellen, was im Platzhalter mit dem Formatierungssymbol `F3` angegeben ist.

### Übung 4

*Schreiben Sie eine Methode, die zwei double-Werte als Parameter übernimmt, beide miteinander multipliziert, das Ergebnis aber als int-Wert zurückliefert. Die Nachkommastellen dürfen einfach abgeschnitten werden.*

Die eigentliche Berechnung ist nicht schwer. Die Konvertierung in den Ergebnistyp realisieren wir durch ein Casting, da ja laut Vorgabe die Nachkommastellen irrelevant sind.

```
public int DoMultiply(double a, double b)
{
  return ((int)(a*b));
}
```

### Übung 5

*Schreiben Sie eine Methode, die zwei als int übergebene Parameter dividiert. Das Ergebnis soll als short-Wert zurückgeliefert werden. Falls die Konvertierung nach short nicht funktioniert, soll das abgefangen werden. Überladen Sie die bestehende Methode zum Dividieren der Werte in den Feldern der Klasse.*

Wir werden also eine zweite Methode mit dem Namen `DoDivide()` schreiben. Da diesmal aber Parameter übergeben werden, kann der Compiler die beiden ganz gut auseinander halten, es sind also keine Probleme zu erwarten.

Wir sollen das Ergebnis einer Division zweier **int**-Werte als **short** zurückliefern. Kein Problem, Casting erledigt die Konvertierung für uns. Und ein **checked**-Block sorgt dafür, dass im Falle eines Konvertierungsfehlers eine Exception ausgelöst wird.

```
public short DoDivide(int a, int b)
{
  checked
  {
    return ((short)(a/b));
  }
}
```

### Übung 6

*Schreiben Sie eine Methode, die zwei* **string**-*Werte zusammenfügt und das Ergebnis als* **string**, *rechtsbündig, mit insgesamt 20 Zeichen, zurückliefert. Erstellen Sie für diese Methode eine eigene Klasse und sorgen Sie dafür, dass die Methode immer verfügbar ist.*

Das Zusammenfügen zweier Strings ist nicht weiter schwer, dafür gibt es mehrere Möglichkeiten. Wir benötigen jedoch eine Möglichkeit, die resultierende Zeichenkette rechtsbündig zu formatieren. Wenn wir in der Tabelle nachschauen, finden wir dort eine Methode `PadLeft()`, die sich hierfür anbietet. Sie füllt einen String von links mit Leerzeichen bis zu einer gewissen Gesamtlänge, die wir angeben können und die in unserem Fall 20 Zeichen beträgt. Außerdem machen wir die Methode noch zu einer statischen Methode, damit sie immer verfügbar ist. Die gesamte Klasse im Zusammenhang:

```
class StringFormatter
{
  public static string DoFormat(string a, string b)
  {
    string c = string.Concat(a,b);
    return (c.PadLeft(20));
  }
}
```

### 12.2.3 Lösungen zu Kapitel 5

#### Übung 1

*Schreiben Sie eine Funktion, die kontrolliert, ob eine übergebene ganze Zahl gerade oder ungerade ist. Ist die Zahl gerade, soll* `true` *zurückgeliefert werden, ist sie ungerade,* `false`.

Zunächst müssen wir uns überlegen, wie wir kontrollieren können, ob eine Zahl gerade oder ungerade ist. Im Prinzip ganz einfach: Eine Zahl ist dann gerade, wenn bei der Division durch 2 kein Rest bleibt. Damit haben wir das einzige benötigte Kriterium bereits vollständig beschrieben. Der Rest der Methode ist trivial.

```
class TestClass
{
  public bool IsEven(int theValue)
  {
    if ((theValue%2)==0)
      return true;
    else
      return false;
  }
}
```

Wenn wir statt der `if`-Anweisung nun den bedingten Zuweisungsoperator verwenden, sieht das Ganze noch einfacher aus:

```
class TestClass
{
  public bool IsEven(int theValue)
  {
    return ((theValue%2)==0)?true:false;
  }
}
```

Am einfachsten wird es aber, wenn man sich überlegt, dass die Bedingung selbst ja bereits einen booleschen Wert zurückliefert. Damit ist auch Folgendes möglich (die wohl einfachste Lösung des Problems):

```
class TestClass
{
  public bool IsEven(int theValue)
  {
    return ((theValue%2)==0);
  }
}
```

### Übung 2

*Schreiben Sie eine Methode, mit der überprüft werden kann, ob es sich bei einer übergebenen Jahreszahl um ein Schaltjahr handelt oder nicht. Ein Jahr ist dann ein Schaltjahr, wenn es entweder durch 4 teilbar, aber nicht durch 100 teilbar ist, oder wenn es durch 4, durch 100 und durch 400 teilbar ist. Die Methode soll* `true` *zurückliefern, wenn es sich um ein Schaltjahr handelt, und* `false`*, wenn nicht.*

Die eigentliche Methode ist nicht weiter schwer zu programmieren, allerdings müssen wir zunächst den richtigen Denkansatz finden. Es soll kontrolliert werden, ob eine übergebene Jahreszahl ein Schaltjahr ist.

Das erste Kriterium ist die Division durch 4. Wenn die übergebene Zahl nicht durch 4 teilbar ist, kann es sich auch nicht um ein Schaltjahr handeln. Wir können also sofort `false` zurückliefern.

Ist die Zahl durch 4 teilbar, müssen wir die Division durch 100 kontrollieren. Ist die Division durch 100 nicht möglich, handelt es sich um ein Schaltjahr – in diesem Fall liefern wir sofort `true` zurück.

Der letzte Fall ist nun der, dass die Zahl durch 4 und durch 100 teilbar ist. Dann besteht das letzte Kriterium in der Division durch 400. Ist diese möglich, handelt es sich um ein Schaltjahr, ansonsten nicht.

Umgesetzt in eine Methode sieht das Ganze so aus:

```
using System;

class Schaltjahr
{
  public bool IsSchaltjahr(int x)
  {
    if ((x%4) != 0)
      return false;
```

```
    if ((x%100) != 0)
      return true;
    if ((x%400) != 0)
      return false;
    return true;
  }
}
```

Auf der CD finden Sie ein funktionierendes Programm basierend auf dieser Klasse im Verzeichnis ÜBUNGEN\KAPITEL_5\SCHALTJAHR.

### Übung 3

*Schreiben Sie eine Methode, die kontrolliert, ob eine Zahl eine Primzahl ist. Der Rückgabewert soll ein boolescher Wert sein.*

Bevor wir uns der eigentlichen Berechnung widmen, sollten wir uns überlegen, wodurch eine Zahl zur Primzahl wird. Primzahlen sind alle Zahlen, die nur durch sich selbst und durch 1 teilbar sind, wobei die 2 keine Primzahl ist. Das bedeutet für uns, dass wir alle Zahlen kleiner 3 zurückweisen müssen. Die eigentliche Kontrolle führen wir mit einer for-Schleife durch.

```
using System;

public class Primzahl
{
  public static bool CheckPrimZahl(int theValue)
  {
    if (theValue<3)
      return(false);

    for (int i=2;i<theValue;i++)
    {
      if ((theValue%i)==0)
        return (false);
    }
    return (true);
  }
}
```

Ein komplettes Programm zur Kontrolle von Primzahlen finden Sie auf der CD im Verzeichnis ÜBUNGEN\KAPITEL5\PRIMZAHL.

### Übung 4

*Schreiben Sie eine Methode, die den größeren zweier übergebener Integer-Werte zurückliefert.*

Diese Methode zu schreiben sollte eigentlich kein Problem darstellen. Wir benutzen der Einfachheit halber eine bedingte Zuweisung, wodurch sich die Funktionalität der Methode auf eine Zeile beschränkt.

```
class TestClass
{
  public int IsBigger(int a, int b)
  {
    return (a>b)?a:b;
  }
}
```

### Übung 5

*Schreiben Sie analog zur Methode ggT auch eine Methode kgV, die das kleinste gemeinsame Vielfache errechnet.*

Wir müssen uns also zunächst Gedanken machen, wie man das kleinste gemeinsame Vielfache errechnen kann. Es handelt sich in jedem Fall um eine ganze Zahl, wie auch beim ggT. Wir wissen auch, dass diese Zahl ohne Rest durch beide vorgegebenen Zahlen dividiert werden kann. Das muss also unser Kriterium sein: Die erste Zahl, die durch beide übergebenen Werte dividiert werden kann, ohne einen Rest zu ergeben, muss die gesuchte Zahl sein.

Den Operator % haben wir bereits bei der ggT-Berechnung benutzt, um zu kontrollieren, ob bei einer Division Nachkommastellen entstehen. Hier werden wir ihn wieder benutzen. Wir programmieren eine **do-while**-Schleife, in der wir einen Wert ständig erhöhen. Sobald dieser Wert unser Kriterium erfüllt, haben wir eine Lösung und verlassen die Schleife. Sie werden sehen, dass wir in diesem Fall nicht einmal eine **break**-Anweisung benötigen – es geht auch ohne.

```
using System;

  public class KGVClass
  {
    public KGVClass()
    {
    }
```

```
  public static int GetKGV(int a, int b)
  {
    int helpVal= a;
    bool isOk = false;
    do
    {
      isOk = (((helpVal%a)==0)&&((helpVal%b)==0));
      if (!isOk)
        helpVal++;
    } while (!isOk);
    return (helpVal);
  }
}

public class TestClass
{
  public static void Main()
  {
    Console.WriteLine("KGV: {0}",KGVClass.GetKGV(5,7));
  }
}
```

Der Einfachheit halber wurde in diesem Beispiel in der `Main()`-Methode nur eine `WriteLine()`-Anweisung benutzt, wobei die für die Berechnung relevanten Werte als feste Werte übergeben wurden. Sie können selbstverständlich auch eine Eingabemöglichkeit vorsehen und dann entsprechend der eingegebenen Werte ein Ergebnis zurückliefern. Ein komplettes Programm mit entsprechender Eingabemöglichkeit finden Sie auf der beiliegenden CD im Verzeichnis ÜBUNGEN\KAPITEL_5\KGV.

### Übung 6

*Erweitern Sie das Beispiel „Quadratwurzel nach Heron“ so, dass keine negativen Werte mehr eingegeben werden können. Wird ein negativer Wert eingegeben, so soll der Anwender darüber benachrichtigt werden und eine weitere Eingabemöglichkeit erhalten.*

Die Berechnung der *Quadratwurzel nach Heron* soll erweitert werden. Es soll nicht möglich sein, negative Werte zu verwenden, bei denen bekanntlich keine Wurzel gezogen werden kann. Nun, zumindest nicht so einfach (betrachtet man sich Differential- und Integralrechnungen, funktioniert es schon).

Da wir dem Anwender eine weitere Eingabemöglichkeit geben wollen, führen wir die Kontrolle auf negative Werte sinnvollerweise in

der Methode Main() durch. Mittels einer Schleife können wir die Eingabe wiederholen lassen, bis der übergebene Wert positiv ist.

```
using System;

class Heron
{
  public double doHeron(double a)
  {
    double A = a;
    double b = 1.0;
    const double g = 0.0004;

    do
    {
      a = (a+b)/2;
      b = A/a;
    }
    while ((a-b)>g);

    return a;
  }
}

class TestClass
{
  public static void Main()
  {
    double x = 0;
    Heron h = new Heron();

    do
    {
      Console.Write("Geben Sie einen Wert ein: ");
      x = Console.ReadLine().ToDouble();
      if (x<=0)
        Console.WriteLine("Der Wert muss >0 sein");
    }
    while (x<=0);

    Console.WriteLine("Wurzel von {0} ist {1}",
                      x,h.doHeron(x));
  }
}
```

Das Programm finden Sie auf der CD im Verzeichnis ÜBUNGEN\KAPITEL_5\HERON.

## 12.2.4 Lösungen zu Kapitel 7

### Übung 1

*Erstellen Sie eine neue Klasse. Programmieren Sie eine Methode, mit deren Hilfe der größte Wert eines Array aus Integer-Werten ermittelt werden kann. Die Methode soll die Position des Wertes im Array zurückliefern.*

Diese Übung ist nicht weiter schwer. Wir kontrollieren die Größe des übergebenen Array, durchlaufen es und merken uns immer den Index des größten Wertes. Diesen liefern wir zurück, nachdem das gesamte Array durchlaufen ist.

```
using System;

class ArrayTest
{
  public static int GetBiggestIndex(int[] theArray)
  {
    int bIndex = 0;

    for (int i=1;i<theArray.Length;i++)
    {
      if (theArray[i]>theArray[bIndex])
        bIndex=i;
    }

    return (bIndex+1);
  }
}
```

Standardmäßig legen wir fest, dass der erste Wert des Array (also Index 0) der größte Wert ist. Danach gehen wir die restlichen Werte durch, und wenn ein größerer Wert auftaucht, merken wir uns den neuen Index in der Variable `bIndex`. Beim Vergleich mit dem nächsten Wert wird ohnehin immer der Wert genommen, der sich an der Stelle, die durch `bIndex` festgelegt ist, befindet. Per Definitionem wird also immer mit dem jeweils größten gefundenen Wert verglichen. Der zurückgelieferte Index ist also zwangsläufig der Index des größten Werts im Array.

Bei der Rückgabe des Werts müssen wir allerdings darauf achten, dass der Anwender als Mensch bekanntlich bei „1“ mit der Zählung beginnt. Daher erwartet er, dass, wenn an der dritten Position im Array der größte Wert zu finden ist, auch der Wert „3“ zurückgeliefert wird.

C# aber beginnt bei 0 mit der Zählung. Wir müssen also dem Ergebnis den Wert 1 hinzuzählen.

Sie finden das gesamte Programm auf der beiliegenden CD im Verzeichnis ÜBUNGEN\KAPITEL_7\ARRAY1.

### Übung 2

*Fügen Sie der Klasse eine Methode hinzu, die die Position des kleinsten Wertes in einem Integer-Array zurückliefert.*

Diese Methode ist natürlich ebenso trivial wie der Vorgänger. Es muss lediglich der Vergleich geändert werden.

```
using System;

class ArrayTest
{
  public static int GetSmallestIndex(int[] theArray)
  {
    int bIndex = 0;

    for (int i=1;i<theArray.Length;i++)
    {
      if (theArray[i]<theArray[bIndex])
        bIndex=i;
    }

    return (bIndex+1);
  }
}
```

Sie finden das gesamte Programm auf der beiliegenden CD im Verzeichnis ÜBUNGEN\KAPITEL_7\ARRAY2.

### Übung 3

*Fügen Sie der Klasse eine Methode hinzu, die die Summe aller Werte des Integer-Array zurückliefert.*

Auch diese Methode ist nicht weiter schwierig. Alle Werte des Array werden zusammengezählt, die Summe zurückgeliefert. Wir gehen der Einfachheit halber von einem **double**-Array aus, natürlich wäre es möglich, die Methode noch zu überladen und damit auch andere Datentypen zu verwenden.

```
using System;

class ArrayTest
{
  public static double ArraySum(double[] theArray)
  {
    double theResult = 0;

    foreach (double x in theArray)
      theResult += x;

    return (theResult);
  }
}
```

Mit der **foreach**-Schleife durchlaufen wir das Array und addieren jeden Wert zur Ergebnisvariable `theResult`. Da wir diese vorher mit 0 initialisiert haben, können wir sicher sein, dass das Ergebnis auch wirklich der Summe aller Werte des Array entspricht.

Sie finden das gesamte Programm auf der beiliegenden CD im Verzeichnis ÜBUNGEN\KAPITEL_7\ARRAY3.

### Übung 4

*Fügen Sie der Klasse eine Methode hinzu, die den Durchschnitt aller Werte im Array zurückliefert.*

Wir sind immer noch in der gleichen Klasse. Für diese Aufgabe können wir auch die Methode komplett übernehmen, wir müssen lediglich eine Anweisung hinzufügen.

```
using System;

class ArrayTest
{
  public static double ArraySchnitt(double[] theArray)
  {
    double theResult = 0;

    foreach (double x in theArray)
      theResult += x
    theResult /= theArray.Length;

    return (theResult);
  }
}
```

Und natürlich funktioniert es auch einfacher, denn die Methode zur Berechnung der Summe aller Werte ist ja bereits vorhanden und wir könnten sie eigentlich auch aufrufen, statt den Code neu zu schreiben. Um Platz zu sparen werden die bereits enthaltenen Methoden nicht nochmals aufgeführt, im Beispiel sind sie aber vorhanden.

```
using System;

class ArrayTest
{
  public static double ArraySchnitt(double[] theArray)
  {
    return (ArraySum(theArray)/theArray.Length);
  }
}
```

Sie finden das gesamte Programm auf der beiliegenden CD im Verzeichnis ÜBUNGEN\KAPITEL_7\ARRAY4.

### 12.2.5 Lösungen zu Kapitel 8

#### Übung 1

*Erstellen Sie eine Basisklasse. Die Klasse soll Personen aufnehmen können, mit Name und Vorname.*

Diese Basisklasse ist schnell implementiert.

```
using System;

class Personen
{
  public string name;
  public string vorname;
}
```

Für diese Klasse sind nicht einmal Methoden notwendig. Auf die Felder Name und Vorname kann nach Erstellung einer Instanz zugegriffen werden.

#### Übung 2

*Leiten Sie zwei Klassen von der Basisklasse ab, eine für männliche Personen, eine für weibliche Personen. Implementieren Sie für die neuen Klassen eine Zählvariable, mit der Sie die Anzahl Männer bzw. Frauen erfassen können.*

```
class Maenner : Personen
{
  public static int theCount = 0;
}

class Frauen : Personen
{
  public static int theCount = 0;
}
```

### Übung 3

*Die Ausgabe des Namens bzw. des Geschlechts soll über ein Interface realisiert werden. Erstellen Sie ein entsprechendes Interface und binden Sie es in die beiden neuen Klassen ein. Sorgen Sie dafür, dass sich die Ausgaben später wirklich unterscheiden, damit eine Kontrolle möglich ist.*

Wir benötigen also zunächst ein Interface, in dem wir die Ausgaberoutine festlegen. Wir benötigen nur diese eine Routine, die wir dann allerdings in der abgeleiteten Klasse implementieren müssen.

```
interface IAusgabe
{
  void doAusgabe();
}

class Maenner : Personen, IAusgabe
{
  public static int theCount = 0;

  public void doAusgabe()
  {
    Console.WriteLine("Herr {0} {1}",this.vorname,this.name);
  }

}

class Frauen : Personen, IAusgabe
{
  public static int theCount = 0;

  public void doAusgabe()
  {
    Console.WriteLine("Frau {0} {1}",this.vorname,this.name);
  }
}
```

Damit ist das Interface bereits implementiert; wenn wir die entsprechenden Methoden aufrufen (nicht ohne ein Casting nach `IAusgabe`), werden die Namen ausgegeben.

### 12.2.6 Lösungen zu Kapitel 9

#### Übung 1

*Erstellen Sie eine Klasse, die einen Notendurchschnitt berechnen kann. Es soll lediglich die Anzahl der geschriebenen Noten eingegeben werden können, damit aber sollen sowohl der Durchschnitt als auch die Gesamtanzahl der Schüler ermittelt werden können. Realisieren Sie alle Felder mit Eigenschaften, wobei die Eigenschaften Durchschnitt und Schüleranzahl nur zum Lesen bereitstehen sollen.*

Jetzt wird es ein wenig umfangreicher, aber nicht besonders kompliziert. Zunächst werden wir die Basisklasse festlegen, mit den Feldern, in denen dann die Werte gespeichert werden.

```
class NotenDurchschnitt
{
  protected int note1;
  protected int note2;
  protected int note3;
  protected int note4;
  protected int note5;
  protected int note6;

  public void doAusgabe()
  {

  }
}
```

Die Methode `doAusgabe()` dient später der Ausgabe aller Werte, falls das gewünscht ist. Variablen für die Schüleranzahl und den Notendurchschnitt benötigen wir auch nicht, denn wir können sie ohnehin nicht zuweisen. Daher werden diese Werte immer anhand der aktuellen Notenzahlen berechnet. Nun müssen wir die Eigenschaften implementieren. Das ist allerdings auch nicht weiter schwer:

```
class NotenDurchschnitt
{
  protected int note1;
  protected int note2;
  protected int note3;
```

```
protected int note4;
protected int note5;
protected int note6;

public int Note1
{
  get { return (note1); }
  set { note1 = value;  }
}

public int Note2
{
  get { return (note2); }
  set { note2 = value;  }
}

public int Note3
{
  get { return (note3); }
  set { note3 = value;  }
}

public int Note4
{
  get { return (note4); }
  set { note4 = value;  }
}

public int Note5
{
  get { return (note5); }
  set { note5 = value;  }
}

public int Note6
{
  get { return (note6); }
  set { note6 = value;  }
}

public int Schueler
{
  get
  {
    return (note1+note2+note3+
            note4+note5+note6);
```

```
    }
  }

  public double Schnitt
  {
    get
    {
      double gesamt = 0;

      gesamt += note1;
      gesamt += (note2*2);
      gesamt += (note3*3);
      gesamt += (note4*4);
      gesamt += (note5*5);
      gesamt += (note6*6);

      return (gesamt/Schueler);
    }
  }

  public void doAusgabe()
  {

  }
}
```

Als Letztes bleibt noch die Ausgabe der Werte. Obwohl diese Methode eigentlich nicht Bestandteil der Übung war, ist es doch sinnvoll, eine zu schreiben.

```
public void doAusgabe()
{
  Console.WriteLine("Anzahl Schüler gesamt: {0}",Schueler);
  Console.WriteLine("\nAnzahl Note 1: {0}",Note1);
  Console.WriteLine("Anzahl Note 2: {0}",Note2);
  Console.WriteLine("Anzahl Note 3: {0}",Note3);
  Console.WriteLine("Anzahl Note 4: {0}",Note4);
  Console.WriteLine("Anzahl Note 5: {0}",Note5);
  Console.WriteLine("Anzahl Note 6: {0}",Note6);
  Console.WriteLine("\nNotendurchschnitt: {0:F2}",Schnitt);
}
```

Ein komplettes Programm finden Sie auf der CD im Verzeichnis ÜBUNGEN\KAPITEL_9\NOTEN.

# A Die Compilerkommandos

Der Kommandozeilencompiler liefert Ihnen eine Liste aller möglichen Parameter bzw. Kommandos, wenn Sie das Fragezeichen bzw. */help* als Parameter eingeben:

```
csc /?
```

oder

```
csc /help
```

Da es aber auf dem Bildschirm immer ein wenig unübersichtlich ist und es sich doch um eine recht große Anzahl von Parametern handelt, hier die Auflistung aller Parameter in Form mehrerer Tabellen.

| **Allgemeine Kommandos** | |
|---|---|
| /? bzw. /help | Zeigt die Kommandos bzw. Parameter an, die möglich sind. |
| /nologo | Unterdrückt die Copyright-Meldung des Compilers. |
| /bugreport:<Dateiname> | Erstellt eine Datei mit einem Fehler-Report. Alle Fehler, Hinweise und Warnungen, die während des Compilerlaufs entstehen, werden in dieser Datei gespeichert. |
| /main:<Klassenname> | Gibt an, welche `Main()`-Methode für den Programmstart benutzt werden soll. Zwar können Sie normalerweise in einem Programm nur eine `Main()`-Methode angeben, mit Hilfe dieses Compilerschalters ist es aber möglich, mehrere `Main()`-Methoden zu programmieren (z.B. für normalen Programmablauf und für einen Debug- oder Testlauf) und die Klasse, deren `Main()`-Methode benutzt werden soll, explizit anzugeben. |

*Tabelle A.1: Allgemeine Compilerkommandos*

| Compilerlauf | |
|---|---|
| /debug[+/-] | Compiliert mit Informationen für den Debugger, zur Fehlersuche. |
| /checked[+/-] | Kontrolliert standardmäßig auf Überlauf bzw. Unterlauf. |
| /unsafe[+/-] | Erlaubt die Verwendung von unsicherem Code im Programm. |
| /d:<Liste> | Definiert Konditionssymbole. |
| /win32res:<Dateiname> | Ermöglicht die Angabe einer Ressourcendatei mit 32-Bit Windows-Ressourcen. |
| /win32icon:<Dateiname> | Ermöglicht die Angabe der Icon-Datei, die für das Programm verwendet werden soll. |
| /res:<Dateiname> | Bettet eine Ressourcendatei in das Projekt mit ein. |

*Tabelle A.2: Kommandos für den Compilerlauf*

| Ausgabeoptionen | |
|---|---|
| /a | Erzeugt eine Anwendung im *Portable Executable-Format*, d.h. im Zwischencode, der dann erst vom .net-Framework compiliert werden muss. |
| /o[+/-] | Schaltet die automatische Optimierung ein oder aus. |
| /out:<Dateiname> | Ermöglicht die Angabe des Dateinamens für die Applikation. |
| /t:<Applikationsart> | Ermöglicht eine Spezifizierung der Art der Applikation. Die Angaben für die Applikationsart können sein: *module* für ein Modul, das zu einer anderen Anwendung hinzugefügt werden kann, *library*, wenn statt einer ausführbaren Datei eine DLL erzeugt werden soll, *exe* wenn es sich um eine Konsolenanwendung handeln soll, und *winexe*, wenn es sich um eine reinrassige Windows-Anwendung mit grafischer Benutzerschnittstelle handeln soll. Standard ist die Erzeugung einer Konsolenanwendung. |
| /nooutput[+/-] | Ermöglicht es, nur den Programmcode auf Korrektheit zu überprüfen. Es wird keine ausführbare Datei erzeugt. |

*Tabelle A.3: Optionen für die Ausgabe*

| Eingabeoptionen | |
|---|---|
| /addmodule:<Dateiname> | Fügt der Anwendung das spezifizierte Modul hinzu. |
| /nostdlib[+/-] | Lässt die Standardbibliothek für Windows außen vor. Diese Option ist nur dann sinnvoll, wenn Sie für ein Betriebssystem mit einer anderen Standard-Bibliothek programmieren. |
| /recurse:<Dateien> | Ermöglicht es dem Compiler, das aktuelle Applikationsverzeichnis rekursiv nach Dateien für die Applikation zu durchsuchen. Damit werden auch die Dateien in den Unterverzeichnissen mit eingebunden. |

*Tabelle A.4: Optionen für die Eingabe*

***weitere Optionen***

Es gibt noch einige weitere Optionen, die hier aufgeführten sind aber die wichtigsten und im Regelfall sollten Sie damit auskommen. Später in diesem Jahr (zumindest hoffen wir das) wird dann die neue Version des Visual Studio .net, und zwar als Vollversion und nicht als Beta, erscheinen und eine einfachere Konfiguration des Compilers ermöglichen. Die Beta können Sie sich bereits jetzt bei Microsoft bestellen, es steht jedem frei, die neue Software zu testen. Der Kostenpunkt liegt dabei sehr niedrig, eigentlich handelt es sich nur um Versand- und Kopierkosten.

# B Tabellen

In diesem Kapitel sind nochmals alle wichtigen Tabellen aufgeführt, damit Sie nicht das ganze Buch durchsuchen müssen, nur um beispielsweise eine Auflistung aller Modifikatoren zu finden.

## B.1 Reservierte Wörter

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

*Tabelle B.1: Die reservierten Wörter von C#*

## B.2 Datentypen

| Alias | Größe | Bereich | Datentyp |
|---|---|---|---|
| sbyte | 8 Bit | -128 bis 127 | `SByte` |
| byte | 8 Bit | 0 bis 255 | `Byte` |
| char | 16 Bit | Nimmt ein 16-Bit-Unicode-Zeichen auf | `Char` |
| short | 16 Bit | -32768 bis 32767 | `Int16` |
| ushort | 16 Bit | 0 bis 65535 | `UInt16` |
| int | 32 Bit | -2147483648 bis 2147483647. | `Int32` |
| uint | 32 Bit | 0 bis 4294967295 | `UInt32` |
| long | 64 Bit | –9223372036854775808 bis 9223372036854775807 | `Int64` |
| ulong | 64 Bit | 0 bis 18446744073709551615 | `UInt64` |
| float | 32 Bit | $\pm 1.5 \times 10^{-45}$ bis $\pm 3.4 \times 10^{38}$ (auf 7 Stellen genau) | `Single` |
| double | 64 Bit | $\pm 5.0 \times 10^{-324}$ bis $\pm 1.7 \times 10^{308}$ (auf 15–16 Stellen genau) | `Double` |
| decimal | 128 Bit | $1.0 \times 10^{-28}$ bis $7.9 \times 10^{28}$ (auf 28–29 Stellen genau) | `Decimal` |
| bool | 1 Bit | `true` oder `false` | `Boolean` |
| string | unb. | Nur begrenzt durch Speicherplatz, für Unicode-Zeichenketten | `String` |

*Tabelle B.2: Die Standard-Datentypen von C#*

## B.3 Modifikatoren

| Modifikator | Bedeutung |
|---|---|
| public | Auf die Variable oder Methode kann auch von außerhalb der Klasse zugegriffen werden. |
| private | Auf die Variable oder Methode kann nur von innerhalb der Klasse bzw. des Datentyps zugegriffen werden. Innerhalb von Klassen ist dies Standard. |
| internal | Der Zugriff auf die Variable bzw. Methode ist beschränkt auf das aktuelle Projekt. |
| protected | Der Zugriff auf die Variable oder Methode ist nur innerhalb der Klasse bzw. durch Klassen, die von der aktuellen Klasse abgeleitet sind, möglich. |
| abstract | Dieser Modifikator bezeichnet Klassen, von denen keine Instanz erzeugt werden kann. Von abstrakten Klassen muss immer zunächst eine Klasse abgeleitet werden. |

*Tabelle B.3: Die Modifikatoren von C#*

| Modifikator | Bedeutung |
|---|---|
| const | Der Modifikator für Konstanten. Der Wert von Feldern, die mit diesem Modifikator deklariert wurden, ist nicht mehr veränderlich. |
| event | Deklariert ein Ereignis (engl. *Event*) |
| extern | Dieser Modifikator zeigt an, dass die entsprechend bezeichnete Methode extern (also nicht innerhalb des aktuellen Projekts) deklariert ist. Sie können so auf Methoden zugreifen, die in DLLs deklariert sind. |
| override | Dient zum Überschreiben bereits implementierter Methoden beim Ableiten einer Klasse. Sie können eine Methode, die in der Basisklasse deklariert ist, in der abgeleiteten Klasse überschreiben. |
| readonly | Mit diesem Modifikator können Sie ein Datenfeld deklarieren, dessen Werte von außerhalb der Klasse nur gelesen werden können. Innerhalb der Klasse ist es nur möglich, Werte über den Konstruktor oder direkt bei der Deklaration zuzuweisen. |
| sealed | Der Modifikator `sealed` versiegelt eine Klasse. Fortan können von dieser Klasse keine anderen Klassen mehr abgeleitet werden. |
| static | Ein Feld oder eine Methode, das/die als `static` deklariert ist, gilt als Bestandteil der Klasse selbst. Die Verwendung der Variable bzw. der Aufruf der Methode benötigt keine Instanziierung der Klasse. |
| virtual | Der Modifikator `virtual` ist sozusagen das Gegenstück zu override. Mit `virtual` werden die Methoden einer Klasse festgelegt, die später überschrieben werden können (mittels `override`). |

*Tabelle B.3: Die Modifikatoren von C# (Forts.)*

## B.4 Formatierungszeichen

| Zeichen | Formatierung |
|---|---|
| C,c | Währung (engl. *Currency*), formatiert den angegebenen Wert als Preis unter Verwendung der landesspezifischen Einstellungen |
| D,d | Dezimalzahl (engl. *Decimal*), formatiert einen ganzzahligen Wert. Die Präzisionszahl gibt die Anzahl der Nachkommastellen an. |
| E,e | Exponential (engl. *Exponential*), wissenschaftliche Notation. Die Präzisionszahl gibt die Nummer der Dezimalstellen an. Bei wissenschaftlicher Notation wird immer mit einer Stelle vor dem Komma gearbeitet. Der Buchstabe „E“ im ausgegebenen Wert steht für „mal 10 hoch“. |
| F,f | Gleitkommazahl (engl. *fixed Point*), formatiert den angegebenen Wert als Zahl mit der durch die Präzisionsangabe festgelegten Anzahl an Nachkommastellen. |

*Tabelle B.4: Zeichen für die Standardformate*

| Zeichen | Formatierung |
|---|---|
| G,g | Kompaktformat (engl. *General*), formatiert den angegebenen Wert entweder als Gleitkommzahl oder in wissenschaftlicher Notation. Ausschlaggebend ist, welches der Formate die kompaktere Darstellung ermöglicht. |
| N,n | Numerisch (engl. *Number*), formatiert die angegebene Zahl als Gleitkommazahl mit Kommas als Tausender-Trennzeichen. Das Dezimalzeichen ist der Punkt. |
| X,x | Hexadezimal, formatiert den angegebenen Wert als hexadezimale Zahl. Der Präzisionswert gibt die Anzahl der Stellen an. Eine angegebene Zahl im Dezimalformat wird automatisch ins Hexadezimalformat umgewandelt. |

*Tabelle B.4: Zeichen für die Standardformate (Forts.)*

| Zeichen | Verwendung |
|---|---|
| # | Platzhalter für eine führende oder nachfolgende Leerstelle. |
| 0 | Platzhalter für eine führende oder nachfolgende 0. |
| . | Der Punkt gibt die Position des Dezimalpunkts an. |
| , | Jedes Komma gibt die Position eines Tausendertrenners an. |
| % | Ermöglicht die Ausgabe als Prozentzahl, wobei die angegebene Zahl mit 100 multipliziert wird. |
| E+0<br>E-0 | Das Auftreten von E+0 oder E-0 nach einer 0 oder nach dem Platzhalter für eine Leerstelle bewirkt die Ausgabe des Wertes in wissenschaftlicher Notation. |
| ; | Das Semikolon wirkt als Trenner für Zahlen, die entweder größer, gleich oder kleiner 0 sind. Die erste Formatierungsangabe bezieht sich auf positive Werte, die zweite auf den Wert 0 und die dritte auf negative Werte. Werden nur zwei Sektionen angegeben, gilt die erste Formatierungsangabe sowohl für positive Zahlen als auch für den Wert 0. |
| \ | Der Backslash bewirkt, dass das nachfolgende Zeichen so ausgegeben wird, wie Sie es in den Formatierungsstring schreiben. Es wirkt nicht als Formatierungszeichen. |
| ' | Wollen Sie mehrere Zeichen ausgeben, die nicht als Teil der Formatierung angesehen werden, können Sie diese in einfache Anführungszeichen setzen. |

*Tabelle B.5: Zeichen für selbst definierte Formate*

## B.5 Operatoren

| Operator | Bedeutung |
|---|---|
| == | Vergleich auf Gleichheit |
| != | Vergleich auf Ungleichheit |
| > | Vergleich auf größer |
| < | Vergleich auf kleiner |
| >= | Vergleich auf größer oder gleich |
| <= | Vergleich auf kleiner oder gleich |

*Tabelle B.6: Vergleichsoperatoren*

| Operator | Bedeutung |
|---|---|
| ! | nicht-Operator (aus `true` wird `false` und umgekehrt) |
| && | und-Verknüpfung (beide Bedingungen müssen wahr sein) |
| \|\| | oder-Verknüpfung (eine der Bedingungen muss wahr sein) |

*Tabelle B.7: Logische Operatoren*

# C C# im Internet

In der heutigen Zeit ist das Internet die größte Informationsquelle. Auch im Bezug auf neue Programmiersprachen ist das so. Aus diesem Grund hier noch einige interessante Links zum Thema C#.

**http://msdn.microsoft.com/NET/default.asp**

Die Homepage des .net-Frameworks, in englischer Sprache.

**http://www.msdn.microsoft.com/vstudio/nextgen/default.asp**

Die Homepage des Visual Studio .net, ebenfalls in englischer Sprache.

**http://www.aspfree.com/quickstart/howto/default.htm**

Alle Beispiele des .net-Frameworks, wiederum in Englisch.

**http://www.csharp-station.com/**

Eine interessante Site in englischer Sprache, mit einem mehrteiligen Tutorial, Artikeln, einer Mailingliste und weiteren Links.

**http://csharpindex.com/**

Ebenfalls eine englische Site, allerdings voll mit Informationen, Artikeln, Tutorials und Links.

**http://www.c-sharpcorner.com/**

Eine weitere Site mit vielen Informationen, Links und Tutorials in englischer Sprache.

**http://csharpindex.com/downloads/textEditors.asp**

Links zu einer großen Anzahl weiterer Editoren, nicht nur für C# sondern auch für andere Sprachen.

**http://csharpindex.com/downloads/tools.asp**

Tools für C#.

Natürlich gibt es noch viele weitere Links, alle aufzulisten ist ohnehin nicht möglich. Die hier angegebene Liste ist aber eine solide Basis für den Start mit C#.

# S Stichwortverzeichnis

## A



## B

## C

## D



## E



## F



## G