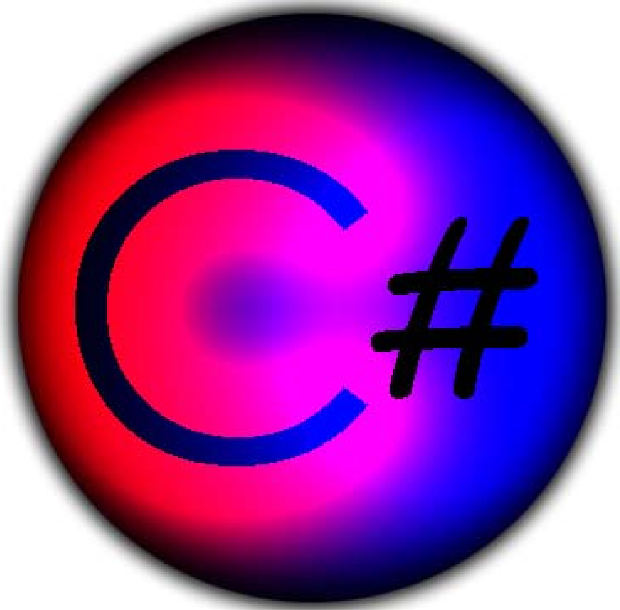
guide to C#



Hinweise zu diesem Dokument

Herkunft des Contents
Der gesamte Content dieses Dokuments stammt von der Internetpräsenz http://www.guidetocsharp.de/ (Datum: 31.Mai 2005)

Copyright
Dort wird eindeutig auf das Copyright hingewiesen:
Copyright 2001-2003 golohaas.de (webmaster@guidetocsharp.de)
Alle Rechte vorbehalten

Genehmigung
Herr Dirk Düsterhöft hat von Herrn Golo Haas die ausdrückliche Genehmigung erhalten, den Content der Internetpräsenz http://www.guidetocsharp.de/ zum Zwecke der Erstellung eines 'eBooks' zu verwenden. Dieses Dokument darf in **unveränderter** Form online und/oder offline verbreitet werden.

Wiedergabe des Contents
Bis auf wenige Stellen wird der Content der Internetpräsenz http://www.guidetocsharp.de/ wörtlich wiedergegeben.
Die Wiedergabe bezieht sich aber nur auf bestimmte Bereiche (s. folgende Baumstruktur):

guide to C#
- **Einführung**
  - **Einführung in C#**
  - **Einführung in .net**
- **Guide**
  - **C#**
    - **Erste Schritte**
    - **Ein- und Ausgabe**
    - **Variablen**
    - **Operatoren**
    - **Entscheidungen**
    - **Schleifen**
    - **Arrays**
    - **Strukturen**
    - **Strings**
    - **Methoden**
    - **Klassen**
    - **Vererbung**
    - **Klassenkonzepte**
    - **Fehlerbehandlung**
    - **Namensräume**
  - **Windows Forms**
    - **Erste Schritte**
    - **Fenster**
    - **Zeichnen**

Schwarz = root **Orange = Ordner, Ebene 1** **Grün = Ordner, Ebene 2** **Blau = Dokument**

Inhaltsverzeichnis

Tabellenverzeichnis ..... *viii*

Codes ..... *viii*

Hinweisboxen ..... *ix*

1 Überblick 1

2 Einführung in C# 2

2.1 Herkunft 2

2.2 Plattformunabhängigkeit 2

2.3 Objektorientiert 3

2.4 Modern 3

2.5 Ausblick 4

3 Einführung in .net 5

3.1 Komponenten von .net 5

3.2 Common language runtime (CLR) 5

3.3 Common language subset (CLS) 7

3.4 Microsoft intermediate language (MSIL) 8

3.5 Windows Forms 9

3.6 ASP.net 10

3.7 ADO.net 11

3.8 XML, SOAP und UDDI 11

3.9 ECMA-Standard 12

3.10 Ausblick 12

4 C# - Erste Schritte 14

4.1 Vorbereitungen 14

4.2 Das erste C#-Programm 15

4.3 Kompilieren 16

4.4 Ausblick 17

5 Ein- und Ausgabe 18

5.1 Ausgabe 18

5.2 Ansi-Escape-Sequenzen 21

5.3 Eingabe 21

5.4 Kommentare 22

5.5 Ausblick 23

6 Variablen **24**
6.1 Einleitung 24
6.2 Deklaration und Definition 26
6.3 Variablennamen 27
6.4 Datentypen 28
6.5 Konvertieren von Variablen 30
6.6 Ausblick 30

7 Operatoren **31**
7.1 Arithmetische Operatoren 31
7.2 Verkürzende Schreibweise 33
7.3 Stringkonkatenationsoperator 36
7.4 Logische Operatoren 36
7.5 Relationale Operatoren 39
7.6 Ausblick 41

8 Entscheidungen **42**
8.1 Einleitung 42
8.2 Die einfache if-Anweisung 42
8.3 Die erweiterte if-Anweisung 44
8.4 Verschachtelte if-Anweisungen 46
8.5 Die else if-Anweisung 48
8.6 Der Bedingungsoperator 49
8.7 Die switch-Anweisung 50
8.8 Ausblick 52

9 Schleifen **53**
9.1 Einleitung 53
9.2 Die for-Schleife 53
9.3 Die break- und die continue-Anweisung 56
9.4 Die while-Schleife 58
9.5 Die do-Schleife 61
9.6 Ausblick 62

10 Arrays **63**
10.1 Einleitung 63
10.2 Arrays 63
10.3 Ein- und Ausgabe der Daten 66
10.4 Sortieren 67

10.5 Referenzdatentypen ... 69
10.6 Arrays mit mehr als einer Dimension ... 70
10.7 Arraygröße ermitteln ... 72
10.8 Die foreach-Schleife ... 73
10.9 Ausblick ... 75

11 Strukturen ... 76
11.1 Einleitung ... 76
11.2 Strukturen ... 76
11.3 Arrays von Strukturen ... 78
11.4 Verschachtelte Strukturen ... 80
11.5 Der Aufzählungsdatentyp enum ... 81
11.6 Ausblick ... 82

12 Strings ... 83
12.1 Vergleichen von Strings ... 83
12.2 Strings kopieren ... 84
12.3 Substrings ... 85
12.4 Strings manipulieren ... 86
12.5 Durchsuchen von Strings ... 87
12.6 Konvertieren von Strings ... 88
12.7 Ausblick ... 89

13 Methoden ... 90
13.1 Einleitung ... 90
13.2 Funktionen ... 90
13.3 Lokale Variablen ... 93
13.4 Parameterübergabe ... 93
13.5 Rückgabewert ... 96
13.6 Call by value / call by reference ... 98
13.7 Die Funktion Main ... 100
13.8 Ausblick ... 102

14 Klassen ... 103
14.1 Einleitung ... 103
14.2 Einführung in Klassen ... 105
14.3 Komplexe Zahlen ... 108
14.4 Konstruktoren ... 114
14.5 Überladen von Methoden ... 116

14.6 *Der this-Zeiger* ... *118*
14.7 *Objekte als Parameter* ... *120*
14.8 *Statische Datenelemente und Methoden* ... *122*
14.9 *Ausblick* ... *124*

15 Vererbung ... 125
15.1 *Vererbung* ... *125*
15.2 *Zugriffsmodifizierer* ... *128*
15.3 *Überschreiben von Methoden* ... *129*
15.4 *Versiegelte und abstrakte Klassen* ... *130*
15.5 *Interfaces* ... *132*
15.6 *Ausblick* ... *133*

16 Klassenkonzepte ... 134
16.1 *Eigenschaften* ... *134*
16.2 *Indexer* ... *136*
16.3 *Delegates* ... *139*
16.4 *Events* ... *142*
16.5 *Überladen von Operatoren* ... *145*
16.6 *Ausblick* ... *148*

17 Fehlerbehandlung ... 149
17.1 *Einleitung* ... *149*
17.2 *Checked und unchecked* ... *150*
17.3 *Ausnahmen abfangen* ... *152*
17.4 *Aufräumcode* ... *156*
17.5 *Ausnahmen auslösen* ... *158*
17.6 *Eigene Ausnahmen definieren* ... *159*
17.7 *Ausblick* ... *159*

18 Namensräume ... 160
18.1 *Einleitung* ... *160*
18.2 *Eigene Namensräume* ... *161*
18.3 *Ausblick* ... *162*

19 Windows Forms - Erste Schritte ... 164
19.1 *Einleitung* ... *164*
19.2 *Windows Forms* ... *164*
19.3 *Hallo Welt GUI* ... *165*

19.4 *Eigene Fenster* ... 170
19.5 *Titelleiste* ... 171
19.6 *Application.Run* ... 172
19.7 *Ausblick* ... 175

20 Fenster ... 176
20.1 *Einleitung* ... 176
20.2 *Form vererben* ... 176
20.3 *Die Main-Methode* ... 177
20.4 *Fenstergröße* ... 178
20.5 *Fensterposition* ... 180
20.6 *Nützliche Eigenschaften* ... 183
20.7 *Ausblick* ... 184

21 Zeichnen ... 185
21.1 *Einleitung* ... 185
21.2 *Ereignisse* ... 185
21.3 *Das Paint-Ereignis* ... 186
21.4 *Zeichnen* ... 187
21.5 *Ungültige Bereiche* ... 189
21.6 *Graphics* ... 190
21.7 *Text ausgeben* ... 191
21.8 *Vererbte Fenster* ... 194

Tabellenverzeichnis

Tabelle 1: Ansi-Escape-Sequenzen 21
Tabelle 2: Datentypen 29
Tabelle 3: Operatoren 32
Tabelle 4: Wetter 37
Tabelle 5: logische Operatoren 38
Tabelle 6: Zustände logischer Operatoren 39
Tabelle 7: Relationale Operatoren 41
Tabelle 8: Rückgabewert von StringCompare() 69
Tabelle 9: Datentypen und ihre Klassen 89
Tabelle 10: call by value / call by reference 99
Tabelle 11: Zugriffsmodifizierer 128
Tabelle 12: get- und set-Funktionen 136
Tabelle 13: Typen von Ausnahmen 156
Tabelle 14: DLLs für Windows Forms 164
Tabelle 15: Werte für Buttons 167
Tabelle 16: Dialog Results 168
Tabelle 17: Buttonwerte und dazugehörige Icons 168
Tabelle 18: Werte für MessageBoxDefaultButton 169
Tabelle 19: Größe und Position von Fenstern 182
Tabelle 20: Rahmenparameter 183

Codes

Code 1: Das erste C#-Programm 15
Code 2: using System 19
Code 3: ReadLine 22
Code 4: Fläche eines Rechtecks berechnen 31
Code 5: Inkrementieren mit Postfix und Präfix 35
Code 6: Wetter 01 37
Code 7: Wetter 02 38
Code 8: Relationale Operatoren 40
Code 9: Einfache if-Anweisung 42
Code 10: Erweiterte if-Anweisung 45
Code 11: Verschachtelte if-Anweisung 47
Code 12: Die else if-Anweisung 49
Code 13: Die switch-Anweisung 51
Code 14: Die for-Schleife 53
Code 15: Quadratzahlen von 1 bis 10 55
Code 16: Die break-Anweisung 57
Code 17: Die continue-Anweisung 58
Code 18: Die while-Schleife 59
Code 19: Die do-Schleife 61
Code 20: Eingabe und Speicherung in einem Array 66
Code 21: Ausgabe von Daten aus einem Array 67
Code 22: Daten ins Array schreiben, sortieren und ausgeben 68
Code 23: Mehrdimensionale Arrays 71
Code 24: Die foreach-Schleife 74
Code 25: Festlegen einer Struktur mittels struct 77
Code 26: Arrays von Strukturen 79

Code 27: Wurzelberechnung durch Aufruf einer Funktion 91
Code 28: Funktionen und Parameterübergabe 94
Code 29: Übergabe mehrerer Argumente an die Funktion Pythagoras 96
Code 30: Modifizierte Funktion Pythagoras 97
Code 31: Die Funktion Main 101
Code 32: Die Funktion Main und erwartete Argumente 101
Code 33: Die Klasse CKomplexeZahl 110
Code 34: Erzeugung von Objekten 112
Code 35: Die Klasse CKomplexeZahl mit Konstruktor 115
Code 36: Die Klasse CKomplexeZahl mit einem zweiten Konstruktor 117
Code 37: Die Klasse CKomplexeZahl um Polymorphismus erweitert 121
Code 38: Statische Datenelemente 123
Code 39: Datenelemente als Eigenschaften implementieren 135
Code 40: Einzelnde Zeichen eines Strings einlesen 137
Code 41: Speichern von IP-Adressen 138
Code 42: Delegates 140
Code 43: Events 144
Code 44: Überladen von Operatoren 147
Code 45: Fakultät berechnen 151
Code 46: Fakultät berechnen mit 'checked' 151
Code 47: Fakultät berechnen mit 'try-/catch-Block' 154
Code 48: Fakultät berechnen mit 'try-/catch- und finally-Block' 158
Code 49: Hallo Welt! mit Windows Forms 165
Code 50: Codeausschnitt zum Abfragen einer Dateispeicherung 169
Code 51: Fenster mit Titelleiste 171
Code 52: Fenster mit Titelleiste und der Methode Run 173
Code 53: Fenster mit Titelleiste und Parameterübergabe 174
Code 54: Anlegen einer Klasse 177
Code 55: Paint-Ereignis 188
Code 56: Weiße Hintergrundfarbe und schwarzer Text 193
Code 57: Vererbung von Fenstern 194
Code 58: Verwenudng der Methode OnPaint 195

Hinweisboxen

Hinweisbox 1: Garbage collection 6
Hinweisbox 2: Neue Klassenbibliothek 6
Hinweisbox 3: Sprachinetgration 8
Hinweisbox 4: Plattformunabhängigkeit 9

1 Überblick[1]

Herzlich willkommen auf guide to C#! Hier finden Sie ein sehr ausführliches Tutorial zu der Programmiersprache C#, das unter anderem durch eine übersichtlich strukturierte Referenz ergänzt wird.

Falls Sie noch nicht genau wissen, was C# ist und sich erst einmal einen allgemeinen Überblick verschaffen möchten, finden Sie nähere Informationen in der Einführung zu C# und .net.

Der Guide hingegen enthält ein in etliche Kapitel unterteiltes Tutorial, in dem Sie zunächst die Sprache C# und später auch die Programmierung mittels .net erlernen können.

Falls Sie sich aber beispielsweise nur über die Syntax eines bestimmten Befehls informieren möchten, können Sie dies mittels der thematisch geordneten Referenz erledigen, die alle Schlüsselwörter von C# ausführlich beschreibt.

In der FAQ finden Sie Antworten auf häufig gestellte Fragen zu den Themen C# und .net. Im Styleguide hingegen finden Sie Tipps und Tricks, um die Qualität und die Lesbarkeit Ihres Codes zu verbessern.

[1] Alle weiteren Hinweise finden sich auf der Internetseite: http://www.guidetocsharp.de
Dies gilt für alle etwaigen Features wie bspw. eine FAQ, die Referenz usw., die in diesem Dokument erwähnt werden.

2 Einführung in C#

Unabhängig davon, ob C# (wird "C Sharp" gesprochen) Ihre erste Programmiersprache ist oder nicht, überlegen Sie sich vielleicht, warum Sie gerade diese Programmiersprache erlernen sollten. Dieser Abschnitt stellt Ihnen daher die Sprache ein bisschen näher vor.

2.1 Herkunft

Die Sprache C# stammt größtenteils von C beziehungsweise C++ ab, enthält allerdings auch Anlehnungen an Java. Dabei hat der Erfinder und Entwickler von C# (Microsoft) versucht, die Sprache so einfach wie möglich und damit weniger anfällig für Fehler zu machen. An C# hat maßgeblich auch der Entwickler der Sprache Delphi, Anders Hejlsberg, mitgearbeitet.

Wenn Sie noch nie mit C oder C++ zu tun hatten, haben Sie vielleicht schon Gerüchte über diese Sprachen gehört, beispielsweise, dass sie schwer zu erlernen und zu beherrschen seien. Lassen Sie sich versichern, dass dies in C# nicht mehr so ist. Viele der Konstrukte, die C und C++ kompliziert machen, gibt es in C# entweder nicht mehr oder sie wurden deutlich vereinfacht.

Wenn Sie schon in C oder C++ programmiert haben, dürfte Sie in diesem Zusammenhang zum Beispiel interessieren, dass es in C# weder Pointer noch Mehrfachvererbung gibt und die drei Operatoren zur Auflösung von Namensbereichen (".", "::" und "->") in einen einzigen zusammengefasst worden sind.

2.2 Plattformunabhängigkeit

Die größte Gemeinsamkeit mit Java dürfte wohl sein, dass die von C# erzeugten Programme plattformunabhängig sind, das heißt, Ihre Programme laufen nicht nur unter dem Betriebssystem, unter dem Sie die Anwendung entwickelt haben, sondern auch auf anderen Systemen, ohne eine einzige Zeile Quellcode anpassen zu müssen.

Um genau zu sein, läuft eine C#-Anwendung auf jeder Plattform, für die das .net Framework, über das Sie im nächsten Kapitel noch etwas erfahren werden, existiert. Nähere Informationen, auf welchen Plattformen .net verfügbar ist, finden Sie im nächsten Kapitel und in der FAQ.

Wenn Sie bereits Programmiererfahrung mit plattformübergreifenden Systemen haben, ist eine wichtige Neuerung von C#, dass seine Datentypen plattformunabhängig sind: So belegt beispielsweise ein Integer immer 32 Bit und hat einen Wertebereich von -2.147.483.648 bis 2.147.483.647, unabhängig von der zugrundeliegenden Hardwareplattform.

2.3 Objektorientiert

C# als moderne Sprache kann natürlich nicht auf Objektorientierung verzichten - erst recht nicht unter Beachtung der Herkunft von C++ und Java. Das besondere an der Objektorientierung von C# ist, dass in C# fast alles als Objekt gesehen wird, selbst die einfachen Datentypen wie int oder long, allerdings ohne die zum Beispiel in Java damit verbundenen Laufzeitverluste mitzubringen.

Des weiteren wird in C# strikt zwischen den Datentypen bool und int differenziert. Es ist somit nicht mehr möglich, in einer if-Anweisung versehentlich eine Zuweisung statt eines Vergleichs durchzuführen, da der Compiler einen Fehler meldet, wenn ein Ausdruck keinen boolschen Wert darstellt.

2.4 Modern

C# ist eine sehr moderne und zukunftssichere Sprache, da es von Microsoft die Sprache der Wahl für .net darstellt. Daher enthält C# viele neue Funktionen, die Sie für die Programmierung moderner Programme benötigen: Von einem neuen, mit einer sehr hohen Genauigkeit arbeitenden Gleitkommadatentyp decimal über eine automatische Ressourcenverwaltung (Garbage collector) bis hin zu einem ausgefeilten System zur Fehler- und Ausnahmebehandlung.

2.5 Ausblick

Nachdem Sie nun einen kleinen Überblick über C# erhalten haben, werden Sie im nächsten Kapitel etwas über den Rahmen, in dem C#-Programme ablaufen, das sogenannte .net Framework erfahren.

3 Einführung in .net

Was ist .net? Auf das notwendigste zusammengefasst, ist .net die neue Vision von Microsoft, auf Informationen und Dienstleistungen jederzeit, an jedem Ort und über jedes beliebige Endgerät zugreifen zu können.

3.1 Komponenten von .net

Dabei besteht .net aus etlichen Einzelkomponenten, unter anderem beispielsweise aus Serverapplikationen (.net Enterprise Server), einer Programmierumgebung (.net Framework) und .net-fähigen Endgeräten (.net Clients).

In diesem Zusammenhang ist für den Entwickler natürlich vor allem die Programmierumgebung, also das .net Framework, interessant. Dieses wird nochmals in etliche Komponenten unterteilt, die im Folgenden vorgestellt werden.

3.2 Common language runtime (CLR)

Den Kern des .net Framework bildet die sogenannte Common language runtime (CLR). Die CLR stellt eine Laufzeitumgebung für Applikationen mit einer äußerst umfangreichen Klassenbibliothek dar.

Die Laufzeitumgebung dient dazu, Applikationen kontrolliert ablaufen zu lassen. Das heißt, die CLR stellt beispielsweise sicher, dass Programme nur Code ausführen, den sie auch ausführen dürfen. So könnte die CLR zum Beispiel Programmen, die aus dem Internet heruntergeladen wurden, verbieten, lokale Daten auszulesen oder zu verändern.

Außerdem beinhaltet die CLR eine Speicher- und Resourcenverwaltung, die Speicher und Resourcen automatisch wieder freigibt, sobald diese nicht länger benötigt werden. Dadurch sinkt einerseits der Aufwand für den Programmierer, andererseits kann es auch nicht mehr passieren, dass nicht benötigter Speicher "aus Versehen" nicht freigegeben wird.

Garbage collection

Das .net Framework enthält eine Garbage collection, die Ressourcen, die nicht mehr benötigt werden, automatisch wieder frei gibt, wie beispielsweise Speicher, Dateien oder Netzwerkverbindungen.

Hinweisbox 1: Garbage collection

Die in der CLR enthaltene Klassenbibliothek ersetzt die bisherige Win32-API sowie deren zahlreichen Erweiterungen wie COM, DCOM oder DNA vollständig. Sie ist komplett objekt-orientiert und wesentlich konsistenter und einfacher aufgebaut als ihre Vorgänger.

So muss man beispielsweise nun nicht mehr zwischen Funktionen, die mittels einer Win32-API-DLL angeboten werden, und Funktionen, die über COM zugänglich sind, unterscheiden.

Ebenso muss man sich nicht mehr mit den unterschiedlichen Vorgehensweisen von Win32 und COM (unter anderem bei der Fehlerbehandlung oder dem Datentyp von Referenzen) auseinander setzen, da diese nun vereinheitlicht sind.

Neue Klassenbibliothek

Das .net Framework enthält eine komplett objektorientierte, konsistente und einfache Klassenbibliothek, die Zugriff auf alle Funktionen des Betriebssystems anbietet.

Hinweisbox 2: Neue Klassenbibliothek

Des weiteren enthält die CLR einige nützliche Features wie beispielsweise Typsicherheit. Dadurch wird während der Laufzeit verhindert, dass ein Programm auf Objekte im Speicher mit einem falschen Datentyp zugreift oder mittels eines zu großen Index auf ein Feld eines Array zugreift, das nicht existiert. Außerdem können Sprünge innerhalb eines Programms nur noch an genau definierte Kontrollpunkte stattfinden.

3.3 Common language subset (CLS)

Eines der größten Probleme von COM ist seine sprachabhängigkeit. Zwar muss man nicht C++ verwenden, um mit COM-Objekten arbeiten zu können - man kann beispielsweise auch Visual Basic benutzen - aber es ist um einiges einfacher. Visual Basic-Programmierer sind unter COM quasi Programmierer "zweiter Klasse".

Das selbe Problem hat auch die Java-API, denn sie ist ebenfalls sprachabhängig, sogar in einem noch weitaus stärkeren Maße als COM. Um die Java-API ansprechen zu können, muss man Java verwenden. Programmierer anderer Sprachen können sie also entweder nicht nutzen oder sie müssen auf Java umsteigen.

Um dieses Problem mit der CLR des .net Frameworks zu vermeiden, hat Microsoft das Common language subset (CLS) definiert. Das CLS definiert eine Menge von Befehlen, die jede Programmiersprache, die Code für .net erzeugen kann, unterstützen muss. Befehle, die nicht im CLS definiert sind, dürfen nicht verwendet werden.

Dadurch sind alle Programmiersprachen, die Code für .net erzeugen können, uneingeschränkt zueinander kompatibel. Das heißt, keine Sprache ist innerhalb des .net Frameworks funktional mächtiger als eine andere. Der Programmierer kann also die Programmiersprache verwenden, die ihm am besten liegt oder die am besten geeignet ist, um ein Problem zu lösen.

Außerdem ermöglicht diese Sprachintegration die Weiterverwendung von Komponenten auch über Sprachgrenzen hinweg. So ist es im .net Framework beispielsweise problemlos möglich, eine Klasse in C++ zu entwickeln, sie in Visual Basic abzuleiten und in C# zu verwenden.

Sprachintegration

Für alle Programmiersprachen, die Code für das .net Framework erzeugen können, gibt es eine von Microsoft definierte Grundmenge an Befehlen, die unterstützt werden muss. Dadurch sind alle .net-Sprachen uneingeschränkt zueinander kompatibel und keine Sprache ist funktional mächtiger als eine andere.

Hinweisbox 3: Sprachinetgration

Um alle Programmiersprachen, die für das .net Framework existieren, zum CLS kompatibel zu machen, sind teilweise sehr umfangreiche Änderungen an der Sprache nötig gewesen.

Beispielsweise schreibt das CLS die Existenz von Klassen und Objekten vor. Da Visual Basic bisher nicht objekt-orientiert war, enthält das neue Visual Basic .net (VB.net) etliche neue Befehle und einige bisherige Befehle sind nicht mehr enthalten, so dass sich bisherige Visual Basic-Entwickler teilweise umstellen müssen.

Von Microsoft selbst gibt es eine Entwicklungsumgebung für .net - Visual Studio .net (VS .net) - , die Compiler für C++ .net, Visual Basic .net, C# .net und JScript .net enthält. Außerdem sind von Drittherstellern etliche .net-Compiler für andere Sprachen in der Entwicklung, von Cobol über Eiffel und Pascal bis hin zu Java.

3.4 Microsoft intermediate language (MSIL)

Außer dem CLS haben alle Programmierpsrachen für das .net Framework noch eine weitere Gemeinsamkeit. Die Compiler erzeugen nämlich keinen direkt ausführbaren Maschinencode, sondern übersetzen die Programme erst in eine plattformunabhängige Zwischensprache namens Microsoft intermediate language (MSIL).

Diese MSIL-Programme werden erst zur Laufzeit mittels eines JIT-Compiler (Just in time) von der CLR in Maschinencode umgewandelt. Dadurch ist ein für das .net Framework entwickeltes Programm plattformunabhängig und ist auf jeder Plattform lauffähig, für die die CLR existiert.

Plattformunabhängigkeit

Compiler für das .net Framework erzeugen keinen Maschinencode, sondern einen plattformunabhängigen Zwischencode (MSIL). Dadurch, dass der MSIL-Code erst bei der Ausführung von der CLR kompiliert wird (JIT-Compiler), sind Applikationen für .net plattformunabhängig.

Hinweisbox 4: Plattformunabhängigkeit

Der JIT-Compiler arbeitet dabei naturgemäß etwas langsamer als ein "echter" Compiler, denn die endgültige Übersetzung findet wie gesagt erst zur Laufzeit statt. Dieser Nachteil wird aber durch zwei Umstände etwas ausgeglichen.

Zum Einen kann der JIT-Compiler spezielle Optimierungen für die jeweilige Zielplattform vornehmen, also beispielsweise MMX, ISSE, 3Dnow!, ..., zum Anderen wird jeder Befehl nur einmal kompiliert und danach in kompilierter Form im Speicher gehalten.

So muss beispielsweise eine Funktion nur bei ihrem ersten Aufruf kompiliert werden, jeder weitere Aufruf verwendet den bereits kompilierten, optimierten und im Speicher bereitgehaltenen Code.

3.5 Windows Forms

Neben diesen internen, abstrakten Komponenten enthält das .net Framework noch weitere, von außen zugängliche Komponenten. Eine dieser Komponenten heißt Windows Forms, die der Erstellung und Gestaltung von grafischen Benutzeroberflächen (GUI) dient.

Windows Forms sind - wie das .net Framework insgesamt - komplett objektorientiert und ersetzen die entsprechenden Funktionen der MFC vollständig. Zudem sind sie (nicht zuletzt auf Grund der Objektorientierung) wesentlich einfacher zu programmieren als die MFC.

Allerdings eignen sich Windows Forms nur zur Entwicklung von grafischen Oberflächen für Applikationen, die lokal auf einem Computer laufen. Grafische, in einem Browser lauffähige Oberflächen für Serverapplikationen (sogenannte Web Forms) lassen sich mit ihnen also nicht entwickeln.

3.6 ASP.net

Dazu dient im .net Framework die Komponente ASP.net, die den Nachfolger der Active Server Pages-Technologie (ASP) darstellt. Mittels ASP.net lassen sich nicht nur - wie bisher - Webseiten durch serverseitigen Skriptcode erweitern, sondern man kann mit ASP.net auch die bereits angesprochenen Web Forms sowie Web Services entwickeln.

Web Forms repräsentieren hierbei Webseiten, die als Frontend für eine auf dem Server laufende Applikation dienen. Das Interessante bei ASP.net ist dabei, dass man für den hinter einer Webseite liegenden Code nicht mehr - wie bei ASP - auf Skriptcode (VBScript) angewiesen ist, sondern diesen in irgendeiner Programmiersprache, für die ein .net-Compiler existiert, schreiben kann.

Dementsprechend kann man auch alle weiteren Vorteile des .net Framework für ASP.net nutzen, das heißt, der Code wird beispielsweise nicht mehr wie bei ASP interpretiert, sondern beim ersten Aufruf kompiliert, was die Ausführungsgeschwindigkeit deutlich steigert. Außerdem lassen sich auch alle Vorteile der CLR nutzen, von der umfangreichen Klassenbibliothek über die gesicherte Ausführung bis hin zur Sprachintegration.

Web Services hingegen repräsentieren die reine Funktionalität einer Webseite, das heißt, sie trennen die Funktionalität vom Design. So ist es mit Web Services beispielsweise problemlos möglich, mittels einer selbst entwickelten Applikation eine Anfrage an einen Suchdienst zu stellen und die Suchergebnisse zurück zu erhalten, ohne das grafische Frontend des Suchdienstes verwenden zu müssen.

Da auch Web Services mittels ASP.net realisiert werden, gelten für sie die gleichen Vorteile wie für Web Forms.

3.7 ADO.net

Die Komponente ADO.net dient dem Zugriff auf Datenbanken und ist der Nachfolger der ADO-Technologie.

ADO.net arbeitet dabei verbindungslos, das heißt, es erfordert keine ständig offene Verbindung zur Datenbank, sondern stellt diese je nach Bedarf für einen kurzen Zeitraum her.

Somit eignet sich ADO.net nicht nur, um klassische Datenbankzugriffe durchzuführen, sondern eignet sich auch besonders für Zugriffe auf Datenbanken, die nur über das Internet verfügbar und eventuell nicht immer erreichbar sind.

3.8 XML, SOAP und UDDI

Nachdem Sie nun die Bestandteile des .net Frameworks kennen, fragen Sie sich vielleicht, wie die Kommunikation zwischen all diesen Komponenten funktioniert.

Hierzu setzt Microsoft nicht auf neue Technologien, sondern nutzt bereits bestehende offene Standards, Protokolle und Datenformate wie beispielsweise XML, SOAP oder UDDI, um .net möglichst einfach in bestehende Systeme integrieren zu können.

Auch die neuen Serverapplikationen von Microsoft, die speziell auf die Zusammenarbeit mit .net ausgerichtet sind (also die .net Enterprise Server) machen extensiv Gebrauch von diesen Standards, allen voran XML als universelles Datenformat.

3.9 ECMA-Standard

Außer der Verwendung von offenen Standards, Protokollen und Datenformaten innerhalb des .net Frameworks ist Microsoft noch einen Schritt weiter gegangen und hat große Teile des .net Frameworks bei der ECMA zur Standardisierung eingereicht.

Das heißt, dass prinzipiell jeder eine eigene Implementierung entwickeln kann, vor allem auch für andere Plattformen als Windows. Teilweise ist hierzu sogar Code von Microsoft verfügbar (Shared source, nähere Informationen hierzu finden Sie in der FAQ).

3.10 Ausblick

Dieses Kapitel konnte bei weitem nicht alle Aspekte von .net abhandeln, aber Sie sollten jetzt zumindest über einen groben Überblick von .net verfügen.

C#

4 C# - Erste Schritte

In diesem Kapitel werden Sie lernen, Ihr erstes, kleines C#-Programm zu schreiben, zu kompilieren und es schließlich auszuführen.

4.1 Vorbereitungen

Zunächst benötigen Sie einen Editor, in dem Sie Ihre Programme entwickeln. Dazu bieten sich mehrere Möglichkeiten an. Prinzipiell reicht jeder einfache Texteditor aus, der Dateien im reinen ASCII-Format abspeichern kann. Unter Windows eignet sich dazu beispielsweise der mitgelieferte Editor namens Notepad, unter Linux ist die Auswahl deutlich größer, hier ist zum Beispiel der Editor nedit sehr gut geeignet.

Mit diesen Editoren lassen sich Programme zwar prinzipiell entwickeln, aber sie bieten keinerlei weiterführende Funktionen wie beispielsweise eine farbliche Hervorhebung von bestimmten Schlüsselwörtern (Syntax highlighting) oder Unterstützung durch Assistenten, die dem Benutzer immer wiederkehrende Routineaufgaben abnehmen können.

Für Entwickler, die entsprechend höhere Ansprüche stellen, gibt es daher auch wesentlich umfangreichere und sehr komfortable Entwicklungs-umgebungen, wie zum Beispiel das sehr gute, kommerzielle Visual Studio .net von Microsoft oder die kostenlose OpenSource-Entwicklungsumgebung SharpDevelop, die ebenfalls sehr empfehlenswert ist.

Wer öfters programmiert, sollte sich unbedingt eine entsprechende Entwicklungsumgebung zulegen da diese durch Ihre Zusatzfunktionen vieles erleichtern und dem Entwickler eine Menge Arbeit ersparen können. Nähere Informationen zu den einzelnen Entwicklungsumgebungen und wie Sie sie erhalten, finden Sie in der FAQ und unter Downloads.

Für den ersten Einstieg reicht ein einfacher Editor aber vollkommen aus. Je nachdem, ob Sie unter Windows oder unter Linux arbeiten, wird der Editor anders gestartet. Im Folgenden wird davon ausgegangen, dass Sie sich mit

Ihrem Betriebssystem so weit auskennen, dass Sie zum Aufruf und der Bedienung Ihres Editors keine Hilfe benötigen.

Für nähere Informationen, wie bestimmte Vorgänge in Ihrer Entwicklungsumgebung (Visual Studio .net, SharpDevelop, ...) funktionieren, schlagen Sie bitte in deren Dokumentation nach.

4.2 Das erste C#-Programm

Ihr allererstes Programm macht nicht viel, es erzeugt lediglich eine Bildschirmausgabe, nämlich `Mein erstes C#-Programm!`. Geben Sie dazu bitte folgenden Quellcode ein (Sie können ihn natürlich auch markieren und per Copy and Paste in den Editor einfügen). Machen Sie sich keine Gedanken, falls Sie den Programmcode nicht auf Anhieb verstehen, er dient vor allem dazu, Ihnen die Schritte zu zeigen, die notwendig sind, um von einem Quellcode im Editor zu einem lauffähigen Programm zu kommen. Die Befehle und Strukturen werden in den weiteren Kapiteln noch ausführlich behandelt.

```
public class ErstesProgramm
{
  public static void Main()
  {
    System.Console.WriteLine("Mein erstes C#-Programm!");
  }
}
```

Code 1: Das erste C#-Programm

Nachdem Sie den Quellcode eingegeben haben, speichern Sie diesen nun bitte unter dem Dateinamen *ErstesProgramm.cs* in einem Verzeichnis Ihrer Wahl ab. Wenn Sie mit dem Windows-Editor Notepad arbeiten, achten Sie bitte darauf, als Dateityp *Alle Dateien* beim Speichern auszuwählen, da sonst automatisch die Endung .txt an den Dateinamen angehängt wird.

4.3 Kompilieren

Als nächstes müssen Sie Ihr Programm kompilieren, damit der Computer es ausführen kann. Beim Kompilieren wird der Quellcode, den Sie eben geschrieben haben und der in einer für Menschen lesbaren Form vorliegt, in einen plattformunabhängigen Code namens MSIL (Microsoft Intermediate Language) übersetzt, so dass Sie diesen Code weitergeben können und ihn sogar - ohne jegliche Anpassung - sowohl unter Windows als auch unter Linux laufen lassen können.

Dieser Zwischencode wird dann, sobald Sie das Programm starten, von einem weiteren Compiler (einem sogenannten JIT-Compiler (Just in time)) in Echtzeit in Maschinensprache übersetzt, die das jeweilige Betriebssystem versteht und ausführen kann. Dieser zweite Übersetzungsvorgang findet jedes Mal statt, wenn Sie das Programm ausführen, so dass Sie das Programm jederzeit wieder auf einer anderen Plattform ohne Anpassung verwenden können.

Dieses Verfahren erscheint zwar zunächst etwas langsamer als die direkte Übersetzung in ausführbare Maschinensprache, hat aber einen Vorteil: Sie müssen sich nämlich beim Kompilieren - anders als bei fast allen anderen Programmiersprachen - nicht festlegen, auf welcher Plattform das Programm später einmal laufen soll.

Genau genommen ist dies kein spezielles Feature von C#, sondern vom .net Framework. Ihr Programm wird nämlich auf allen Plattformen laufen, für die das .net Framework verfügbar ist, also unter anderem auf Windows, Linux und MacOS X.

Um Ihr Programm also nun zu kompilieren (nähere Informationen, wie Sie den Compiler erhalten, finden Sie wiederum in der FAQ und unter Downloads), öffnen Sie bitte ein DOS-Fenster beziehungsweise ein Terminalfenster. Wechseln Sie nun an der Kommandozeile in das Verzeichnis, in das Sie Ihr Programm abgespeichert haben.

Sofern Sie unter Windows arbeiten, geben Sie bitte folgenden Befehl ein:

```
csc ErstesProgramm.cs
```

Sofern Sie unter Linux arbeiten, lautet der Befehl:

```
mcs ErstesProgramm.cs
```

Diese beiden Befehle bewirken jeweils, dass das Programm kompiliert und somit in die Zwischensprache MSIL übersetzt wird. Sofern Sie alles richtig gemacht haben und Ihr Programm keinen Tippfehler enthält, finden Sie nun im aktuellen Verzeichnis eine Datei namens ErstesProgramm.exe vor, das Sie unter Windows durch Aufruf von `ErstesProgramm` und unter Linux durch Aufruf von `mono ErstesProgramm` starten können.
Auf dem Bildschirm müsste jetzt die Meldung `Mein erstes C#-Programm!` erscheinen. Haben Sie beim Kompilieren eine Fehler-meldung erhalten, schauen Sie sich bitte den Quellcode noch einmal genau an, vielleicht enthält er einen Tippfehler.

4.4 Ausblick

Im nächsten Kapitel werden Sie lernen, wie man Daten auf dem Bildschirm ausgibt und wie man Daten von der Tastatur einliest.

5 Ein- und Ausgabe

Dieses Kapitel befasst sich mit den verschiedenen Möglichkeiten, Bildschirmausgaben zu erzeugen und dem Einlesen von Daten über die Tastatur.

5.1 Ausgabe

Ein Beispiel für eine Bildschirmausgabe haben Sie bereits im letzten Kapitel kennengelernt. Im Folgenden ist die für die Ausgabe verantwortliche Zeile noch einmal aufgeführt:

```
System.Console.WriteLine("Mein erstes C#-Programm!");
```

Wie man leicht sehen kann, steht der Text, welcher ausgegeben werden soll, in Anführungszeichen in den runden Klammern. Die Anführungszeichen sind notwendig, damit C# die einzelnen Wörter als zusammenhängenden Text erkennt. Außerdem ist der Text ein sogenannter Parameter. Parameter stehen immer in den runden Klammern hinter einem Befehl. Der eigentliche Befehl, welcher für die Bildschirmausgabe verantwortlich ist, ist

```
System.Console.WriteLine.
```

WriteLine beschreibt hierbei die Aktion (in unserem Fall also die Ausgabe), die auf die Console (also die Ein- und Ausgabegeräte) ausgeführt wird. Die Console gehört hierarchisch zum System. Solche Hierarchien werden in C# durch einen . gekennzeichnet. Schließlich folgt in C# fast jedem Befehl ein Semikolon, um den Befehl abzuschließen.

Da die meisten Programme sehr viele Ausgaben enthalten, ist es lästig, vor jedem `WriteLine` erst `System.Console.` schreiben zu müssen. Daher gibt es eine Technik, dies zu umgehen:

Man fügt als allererste Zeile (also noch vor der Zeile, die mit `public class` beginnt!) in das Programm folgendes ein:

```
using System;
```

Dann sieht der Quellcode folgendermaßen aus:

```
using System;

public class ErstesProgramm
{
  public static void Main()
  {
    Console.WriteLine("Mein erstes C#-Programm!");
  }
}
```

Code 2: using System

Damit kann man sich zumindest bei jedem `WriteLine` das `System.` ersparen, was vor allem bei längeren Programmen eine große Erleichterung ist. Allerdings darf man trotzdem weiterhin die vollständige Syntax benutzen.

Wie Sie vielleicht schon selbst entdeckt haben (wenn Sie mehrere `WriteLine`-Befehle hintereinander ausgeführt haben), bewirkt der Befehl `WriteLine`, dass nach der Ausgabe des Textes ein Zeilenumbruch stattfindet. Manchmal möchte man dies jedoch nicht. Um es zu verhindern, kann man alternativ den Befehl `Write` verwenden.

```
Console.WriteLine("Halli-");
Console.WriteLine("Hallo");
```

erzeugt beispielsweise

```
Halli-
Hallo
```

Dagegen gibt

```
Console.Write("Halli-");
Console.Write("Hallo");
```

den Text

```
Halli-Hallo
```

auf dem Bildschirm aus.

Manchmal möchte man aber auch mehrere kurze Texte jeweils in einer neuen Zeile ausgeben. Um nicht für jede Zeile ein neues `WriteLine` verwenden zu müssen, gibt es noch eine andere Möglichkeit, einen Zeilenumbruch zu erzeugen: Man fügt einfach `\n` an der Stelle in den Text ein, wo der Zeilenumbruch stattfinden soll.

Ein Beispiel:

```
Console.Write("Halli-\nHallo");
```

erzeugt die selbe (zweizeilige) Ausgabe wie das Beispiel mit WriteLine. Selbstverständlich kann man auch mehrere `\n` aneinanderfügen, um zwischen zwei Textzeilen mehrere Leerzeilen zu erzeugen.

Alternativ kann man auch durch Aufruf von

```
Console.WriteLine();
```

eine Leerzeile erzeugen.

5.2 Ansi-Escape-Sequenzen

Zeichenfolgen wie `\n` werden als sogenannte Ansi-Escape-Sequenzen bezeichnet. Einige davon und ihre Funktion sind in der folgenden Tabelle aufgelistet:

Ansi-Escape-Sequenz	*Funktion*
\'	Einfaches Hochkomma
\"	Anführungszeichen
\\	Umgekehrter Schrägstrich (Backslash)
\0	Null (Unicode-Wert 0)
\a	Piepton (Alert)
\b	Rückschritt (Backspace)
\f	Seitenvorschub (Form feed)
\n	Neue Zeile (New line)
\r	Wagenrücklauf (Carriage return)
\t	Tabulator
\v	Vertikaler Tabulator

Tabelle 1: Ansi-Escape-Sequenzen

5.3 Eingabe

Oft möchte man nicht nur feste Daten auf dem Bildschirm ausgeben, sondern benötigt (zum Beispiel zur Berechnung von Daten) noch weitere Informationen vom Benutzer.

Dazu steht Ihnen in C# der Befehl `System.Console.ReadLine()`zur Verfügung. Um seine Funktionsweise zu demonstrieren, hier ein kleines Beispiel (s. Folgeseite):

```
using System;

public class Name
{
  public static void Main()
  {
    Console.Write("Bitte geben Sie Ihren Namen ein: ");
    string Name;
    Name = Console.ReadLine();
    Console.WriteLine("Aha, sie heißen also " + Name + "!");
  }
}
```

Code 3: ReadLine

Dieses Programm bittet Sie, Ihren Namen einzugeben und gibt dann einen Antwortsatz auf dem Bildschirm aus. Den Anfang der Zeile, die mit `string` beginnt, müssen Sie noch nicht verstehen, er wird aber im nächsten Kapitel Variablen erklärt.

5.4 Kommentare

Sie können in Ihre Programme Kommentare einfügen, um Ihren Code zu erläutern. Kommentare werden von Anfängern oft als überflüssig und als Zeitverschwendung angesehen, aber Sie sollten sich trotzdem angewöhnen, die prinzipielle Funktionsweise Ihres Codes zu kommentieren.

Sonst kann es Ihnen bei komplexeren Programmen passieren, dass Sie nach einiger Zeit, wenn Sie die Arbeit an einem älteren Programm wieder aufnehmen, Ihren Code nicht mehr verstehen. Und nichts ist ärgerlicher, als eine Menge Zeit in das Herausfinden und Nachvollziehen der Funktionsweise von selbst geschriebenem Code zu investieren.

Kommentare können Sie in C# gleich auf zwei verschiedene Arten setzen, entweder mittels des Ausdrucks `//` oder mittels des Ausdruckspaares `/*` und `*/`. Die erste Variante kennzeichnet einfach alles in dieser Zeile, was folgt, als Kommentar, in den meisten Fällen sogar eine ganze Zeile:

```
// Diese Zeile ist ein Kommentar

Console.WriteLine("Hallo!");   // Ab hier ist diese Zeile
ein Kommentar

// Console.WriteLine("Hallo!");   Hier ist die ganze
Zeile ein Kommentar
```

Um mehrzeilige Kommentare zu setzen, muss man entweder entsprechend oft `//` benutzen oder man wählt die zweite Syntax für Kommentare. Hier wird alles, was sich zwischen `/*` und `*/` befindet, als Kommentar gewertet, egal über viele Zeilen der Kommentar geht.

```
/* Hier beginnt ein Kommentar,
der über mehr als eine Zeile
geht! */
```

Sie müssen lediglich aufpassen, wenn Sie mehrzeilige Kommentare setzen, dass der Bereich keinen weiteren mehrzeiligen Kommentar enthält, denn C# interpretiert das erste auftauchende Zeichen `*/` als Ende des gesamten Kommentars (siehe Folgeseite)!

```
/* Hier beginnt der äußere Kommentar
Console.WriteLine("Wird nicht ausgeführt!");
     /* Hier beginnt der innere Kommentar
     Console.WriteLine("Wird auch nicht ausgeführt!");
     */
Console.WriteLine("Die vorige Zeile schließt den
äußeren!");
*/   // Diese Zeile erzeugt einen Fehler
```

5.5 Ausblick

Im nächsten Kapitel lernen Sie, was Variablen sind, welche Variablentypen es in C# gibt und was man mit Ihnen machen kann.

6 Variablen

In diesem Kapitel werden Sie lernen, was Variablen und Datentypen sind und wie man in C# mit Ihnen umgeht.

6.1 Einleitung

Am Schluss des letzten Kapitels haben Sie ein kleines Programm geschrieben, welches Sie auffordert, Ihren Namen einzugeben und Sie dann persönlich begrüßt. In diesem Programm sind Sie bereits mit einer ersten Variable in Berührung gekommen. Hier noch einmal der Quellcode:

```
using System;

public class Name
{
  public static void Main()
  {
    Console.Write("Bitte geben Sie Ihren Namen ein: ");
    string Name;
    Name = Console.ReadLine();
    Console.WriteLine("Aha, sie heißen also " + Name + "!");
  }
}
```

(vgl. Code3: ReadLine)

Eine Variable ist eine Art Platzhalter für Daten, in dem Daten gespeichert und wieder abgerufen werden können. Wie Sie sich nun vielleicht schon denken können, heißt die Variable in unserem Fall *Name*.

```
Console.WriteLine("Aha, sie heißen also " + Name + "!");
```

In dieser Zeile wird die Information, die in der Variablen *Name* gespeichert ist, ausgelesen und in den Text eingefügt.
Da Sie den Inhalt der Variablen ausgeben wollen und nicht den Text *Name*, darf *Name* nicht in Anführungszeichen stehen, sondern muss zwischen die beiden - durch Anführungszeichen abgeschlossenen - Texte mittels eines + eingefügt werden.

Dort, wo im Quellcode *Name* steht, wird also zur Laufzeit des Programms der Name des Anwenders eingefügt und ausgegeben. Damit aber überhaupt ein Name ausgegeben werden kann, muss die Variable *Name* erst einmal eine Information (den sogenannten Wert einer Variablen) enthalten. Dies geschieht mittels der Zeile

```
Name = Console.ReadLine();
```

Mittels `Console.ReadLine` wird ein Text von der Tastatur gelesen und dann in der Variablen *Name* gespeichert. Der Wert einer Variablen kann aber auch direkt im Programm zugewiesen werden, wenn ein Wert bereits feststeht. Beispielsweise weist die Zeile

```
Text = "Hallo Welt!";
```

der Variablen Text den Wert *Hallo Welt!* zu. Erwähnenswert ist noch, dass es in C# eine verkürzende Syntax gibt, um mehreren Variablen den selben Wert zuzuweisen, statt

```
a = 5;
b = 5;
```

kann man auch

```
a = b = 5;
```

schreiben. Allerdings sollten Sie diese verkürzende Syntax nach Möglichkeit nicht verwenden, da sie nicht sehr verbreitet ist und andere Programmierer eher verwirren könnte.

6.2 Deklaration und Definition

Jetzt ist nur noch eine Zeile aus dem Programm erklärungsbedürftig, nämlich:

```
string Name;
```

Damit wird C# bekannt gegeben, welche Art von Daten in der Variablen *Name* gespeichert werden, in unserem Fall ein `string` (also ein Text). Dieses Bekanntgeben des Typs einer Variablen nennt man übrigens Deklaration. Es gibt noch viele andere Datentypen (für Ganzzahlen, Kommazahlen, einzelne Zeichen, ...), die Sie noch kennen lernen werden.

Manchmal steht schon bei der Deklaration einer Variablen der Wert fest, den sie später einmal haben soll. Dann kann man die Deklaration um eine Wertzuweisung erweitern. Statt

```
string Text;
Text = "Hallo Welt!";
```

kann man auch kürzer

```
string Text = "Hallo Welt!";
```

schreiben. Fasst man die Deklaration mit einer Wertzuweisung zusammen, so nennt man dies Definition. Variablen in C# sind - sofern sie nur deklariert werden - übrigens nicht automatisch mit einem Standardwert versehen. Sie müssen also einer Variablen einen konkreten Wert zuweisen, sonst wird sich der Compiler beschweren.

Wenn Sie mehrere Variablen des selben Typs anlegen wollen, können Sie die benötigten Anweisungen übrigens auch in eine Zeile zusammen fassen, indem Sie die Variablennamen einfach durch Kommata trennen.

```
string Vorname, Nachname;
```

macht demnach dasselbe wie

```
string Vorname;
string Nachname;
```

Sie können bei dieser Kurzschreibweise Variablen sogar Werte zuweisen.

```
string Vorname = "Bill", Nachname = "Gates";
```

Allerdings ist diese Schreibweise meistens auch etwas unübersichtlicher und wird daher hier nur selten verwendet.

6.3 Variablennamen

Es ist Ihnen - bis auf einige wenige Einschränkungen - übrigens freigestellt, wie Sie die Variablen in Ihrem Programm benennen. Variablennamen dürfen aus folgenden Zeichen bestehen: Groß- und Kleinbuchstaben, Zahlen und dem Zeichen _ (wobei Zahlen aber nicht am Anfang des Variablennamens stehen dürfen). Allerdings müssen Sie auf die Groß-/Kleinschreibung achten. So bezeichnen beispielsweise text und Text unterschiedliche Variablen. Außerdem sollten Sie Umlaute und sprachspezifische Sonderzeichen vermeiden.

Folgende Variablennamen sind also beispielsweise legitim:

```
text
Text
TEXT
Text_und_Zahlen
Text123
_Text_123
```

Nicht legitim sind dagegen zum Beispiel folgende:

```
123_Text
123
Text!
Text und Zahlen
```

Allerdings gibt es in C# eine Vorgabe bezüglich der Groß-/Kleinschreibung von Variablennamen, an die man sich zwar nicht halten muss, aber es erhöht die Lesbarkeit des Codes zum einen für den Autor selbst als auch für andere Programmierer, da diese sich an die selben Regeln halten.

Laut Vorgabe muss jede Variable klein geschrieben werden, mit Ausnahme des Anfangsbuchstaben und der Anfangsbuchstaben von weiteren Wörtern bei aus mehreren Wörtern zusammengesetzten Variablennamen. Ein paar Beispiele für Variablennamen, die der C#-Vorgabe entsprechen:

```
Summe
RestVonDivision
Zieldatei
ZustandDesBildschirms
```

6.4 Datentypen

Sie wissen jetzt, wie man Variablen deklariert und ihnen einen Wert zuweist. Allerdings haben Sie bisher nur mit Variablen des Typs `string` gearbeitet. Doch es gibt in C# noch viel mehr Datentypen (s. folgende Tabelle 2):

Datentyp	Wertebereich	Speicher
Ganzzahlen		
byte	0 bis 255	8 Bit
sbyte	-128 bis 127	8 Bit
short	-32.768 bis 32.767	16 Bit
ushort	0 bis 65.535	16 Bit
int	-2.147.483.648 bis 2.147.483.647	32 Bit
uint	0 bis 4.294.967.295	32 Bit
long	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807	64 Bit
ulong	0 bis 18.446.744.073.709.551.615	64 Bit
Gleitkommazahlen		
float	$1.5 * 10^{-45}$ bis $3.4 * 10^{38}$	32 Bit
double	$5.0 * 10^{-324}$ bis $1.7 * 10^{308}$	64 Bit
decimal	$1.0 * 10^{-28}$ bis $7.9 * 10^{28}$	128 Bit
Zeichen		
char	Einzelne Zeichen	16 Bit
string	Zeichenketten	-
Sonstiges		
bool	true und false	1 Bit

Tabelle 2: Datentypen

Wie Sie sehen, gibt es für fast jede Aufgabe mehrere Datentypen zur Auswahl, für Ganzzahlen sogar acht verschiedene. Prinzipiell gilt, dass ein Datentyp desto mehr Speicher belegt, je größer sein Wertebereich ist beziehungsweise je präziser er Nachkommastellen darstellt. In der Praxis versucht man daher immer, den optimalen Kompromiss zwischen Speicherbelegung und gewünschtem Wertebereich zu finden.

6.5 Konvertieren von Variablen

Manchmal muss man den Datentyp einer Variablen während des Programmablaufs ändern, beispielsweise wenn der Benutzer einen Integer-Wert eingibt, das Programm aber einen Double-Wert erwartet.

Sofern bei der Konvertierung keine Daten verloren gehen können, nimmt C# diese Konvertierung implizit vor. Von der Konvertierung von `int` nach `double` beispielsweise bemerkt man gar nichts, da der Wertebereich von `double` größer als der von `int` ist.

Manche Konvertierungen sind aber nicht so offensichtlich, oder sie gehen aus dem Kontext nicht unbedingt hervor. Beispielsweise könnte man ein `char` nach `int` konvertieren wollen, um so den ASCII-Code des Zeichens zu erhalten. Dann muss man C# explizit anweisen, eine Konvertierung durchzuführen, indem man den Zieldatentyp in runden Klammern vor die Variable setzt.

Das folgende Programmfragment nimmt eben diese Konvertierung vor:

```
char Zeichen = 'A';
Console.WriteLine((int) Zeichen);    // 65
```

Allerdings gibt es einige wenige Konvertierungsfälle, die durch diese Syntax nicht abgedeckt werden. So kann man auf diese Art und Weise beispielsweise keinen `string` in einen anderen Wert konvertieren. Wie Strings konvertiert werden, werden Sie später im Kapitel *Strings* erfahren.

6.6 Ausblick

Im nächsten Kapitel werden Sie sogenannte Operatoren kennen lernen, um mit Variablen zu rechnen oder Strings zu verknüpfen.

7 Operatoren

In diesem Kapitel werden Sie lernen, was Operatoren sind und wie man mit ihnen die Werte von Variablen manipulieren kann.

7.1 Arithmetische Operatoren

Nachdem Sie im letzten Kapitel gelernt haben, was Variablen sind, können Sie nun Daten in Ihren Programmen speichern und wieder ausgeben. Allerdings sind die Daten noch statisch, das heißt, Sie können sie nicht verändern, außer einer Variablen einen neuen Wert zuzuweisen.

Doch selbst dabei haben Sie bereits einen ersten Operator benutzt, ohne es zu wissen: Den Zuweisungsoperator `=`. Nun sollen Sie einige weitere Operatoren kennenlernen.

Die einfachste Veränderung, die Sie an Variablenwerten vornehmen können, sind Berechnungen. Welche Berechnungen durchgeführt werden sollen, bestimmen Sie mittels Operatoren. Dies soll an einem einfachen Beispiel gezeigt werden:

```
using System;

public class Rechnen
{
  public static void Main()
  {
    int Laenge = 30;
    int Breite = 20;

    int Flaeche = Laenge * Breite;

    Console.WriteLine("Die Fläche beträgt {0} Einheiten!",
Flaeche);
  }
}
```

Code 4: Fläche eines Rechtecks berechnen

Dieses Programm definiert zwei Variablen *Laenge* und *Breite*, um ein Rechteck zu beschreiben, und berechnet die Fläche des Rechtecks, indem die beiden Variablen *Laenge* und *Breite* miteinander multipliziert werden. Die Multiplikation wird in C# durch den Operator * durchgeführt.

Da dieser Operator eine arithmetische Operation (also eine Rechen-operation) durchführt, wird er als arithmetischer Operator bezeichnet. Es gibt in C# aber noch mehr arithmetische Operatoren:

Operator	***Funktion***	***Beispiel***

+	Addition	int Summand1 = 5;
		int Summand2 = 7;
		int Summe = Summand1 + Summand2;
		
		Die Variable *Summe* enthält den Wert 12.

-	Subtraktion	int Summand1 = 5;
		int Summand2 = 7;
		int Differenz = Summand1 - Summand2;
		
		Die Variable *Differenz* enthält den Wert -2.

*	Multiplikation	int Faktor1 = 5;
		int Faktor2 = 7;
		int Produkt = Faktor1 * Faktor2;
		
		Die Variable *Produkt* enthält den Wert 35.

/	Division	int Dividend = 35;
		int Divisor = 7;
		int Quotient = Dividend / Divisor;
		
		Die Variable *Quotient* enthält den Wert 5.

%	Modulo-Division	int Dividend = 35;
		int Divisor = 8;
		int Rest = Dividend % Divisor;
		
		Die Variable *Rest* enthält den Wert 3 (35 / 8 = 4, Rest 3).

Tabelle 3: Operatoren

Die Verwendung der arithmetischen Operatoren ist offensichtlich sehr einfach. Im Prinzip verhält sich alles so, wie Sie es von der Mathematik kennen. C# hält selbstverständlich auch die mathematischen Regeln wie beispielsweise Punkt-vor-Strich-Rechnung ein.

Um bei komplexen Ausdrücken (wenn Sie also mehrere Operatoren miteinander kombinieren) die Übersicht zu behalten, können Sie Ausdrücke klammern. Dabei werden Ausdrücke, die weiter innen stehen, zuerst berechnet. So ergibt beispielsweise die Rechnung

```
(1 + 2) * 3
```

den Wert 9, da zuerst der Wert in der Klammer berechnet wird, und dieser dann mit 3 multipliziert wird, während

```
1 + 2 * 3
```

den Wert 7 ergibt, da hier zuerst die Multiplikation ausgeführt wird und dann das Ergebnis der Multiplikation um eins erhöht wird.

7.2 Verkürzende Schreibweise

Oft kommt es vor, dass man eine Variable lediglich um einen bestimmten Wert verändern will, ohne das Ergebnis in einer anderen Variablen zu speichern. Zum Beispiel:

```
i = i + 5;
```

oder

```
i = i * 2;
```

Um bei solchen Ausdrücken die Variable nicht immer doppelt hinschreiben zu müssen, gibt es in C# eine verkürzende Schreibweise. So kann man statt

```
i = i + 5;
```

auch

```
i += 5;
```

schreiben. Beides hat den gleichen Effekt, nämlich dass der Wert der Variablen *i* um 5 erhöht wird. Diese Verkürzung können Sie mit jedem arithmetischen Operator durchführen, es existieren also auch `-=`, `*=`, `/=` und `%=`. Auch wenn Ihnen die Schreibweise anfangs etwas ungewohnt erscheint, lohnt es sich, sie sich anzugewöhnen.

Eine Anweisung, die in Programmen sehr häufig anzutreffen ist, ist die Erhöhung oder Verminderung einer Variablen um den Wert 1. Dank der eben gelernten Verkürzung kann man statt

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

auch

```
i += 1;
```

schreiben. Doch da die Veränderung um 1 so häufig auftritt, gibt es hierfür eine weitere Verkürzung:

```
i++;
```

für die Erhöhung beziehungsweise

```
i--;
```

für die Verminderung. Die Operatoren `++` und `--` heißen Inkrement- beziehungsweise Dekrementoperator. Allerdings muss man bei ihrer Anwendung etwas beachten: Es gibt von beiden nämlich eine sogenannte Präfix-Schreibweise (dann steht der Operator vor der Variablen) und eine Postfix-Schreibweise (dann steht der Operator hinter der Variablen), die sich in der Wirkung leicht unterscheiden.

Die Postfix-Schreibweise bewirkt, dass die Variable erst verändert wird, nachdem andere Berechnungen oder Zuweisungen mit ihr ausgeführt wurden, während die Präfix-Schreibweise zuerst den Wert verändert.

```
using System;

public class Inkrement
{
  public static void Main()
  {
   int i = 27;

   // Die ersten drei Beispiele sind klar

    Console.WriteLine(i);     // 27
    i++;
    Console.WriteLine(i);     // 28
    ++i;
    Console.WriteLine(i);     // 29

   // Nun wird es komplizierter

   Console.WriteLine(i++); // auch 29, aber i enthält am Schluss den
                           // Wert 30, da erst erhöht wird, wenn die
                           // andere Anweisung abgearbeitet ist
   Console.WriteLine(++i); // 31, da i erhöht wird, bevor die andere
                           // Anweisung ausgeführt wird
  }
}
```

Code 5: Inkrementieren mit Postfix und Präfix

7.3 Stringkonkatenationsoperator

Der Operator `+` hat außer der arithmetischen noch eine weitere Funktion in C#, die Sie im letzten Kapitel schon kennen gelernt haben. Er dient nämlich der Konkatenation von Strings, das heißt der Aneinanderreihung.

Im ersten Programmbeispiel des letzten Kapitels haben Sie die Zeile

```
Console.WriteLine("Aha, sie heißen also " + Name + "!");
```

verwendet, um die beiden Stringkonstanten `Aha, sie heißen also` und `!` mit dem String, der in der Variablen Name gespeichert ist, zu verbinden. Dieses Verbinden wird als Konkatenation bezeichnet.

7.4 Logische Operatoren

Neben den arithmetischen gibt es in C# noch einen weiteren Typ von Operatoren: Die logischen Operatoren. Um zu verstehen, wie logische Operatoren funktionieren, muss man sich zunächst noch einmal den Datentyp `bool` vor Augen führen.

Im letzten Kapitel haben Sie gelernt, dass die einzigen Werte, die eine Variable des Typs `bool` annehmen kann, die beiden Literale `true` und `false` sind. Den hauptsächlichen Einsatzbereich einer solchen Variablen werden Sie zwar erst im nächsten Kapitel lernen, aber ein paar Grundlagen werden Sie jetzt schon erfahren.

Stellen Sie sich vor, Sie wollen mittels eines Programms beschreiben, wie das Wetter ist. Sie könnten dazu eine Variable anlegen, in der gespeichert wird, ob die Sonne scheint, und eine weitere, in der gespeichert wird, ob es windig ist. Beide Variablen sind vom Typ `bool`, denn sie können beide entweder nur eine wahre oder eine falsche Aussage beinhalten: Entweder scheint die Sonne (`true`) oder sie scheint nicht (`false`), entweder ist es windig (`true`) oder es nicht windig (`false`).

```
public class Wetter
{
  public static void Main()
  {
    bool SonneScheint = true;
    bool WindWeht = false;
  }
}
```

Code 6: Wetter 01

In diesem Programm geht man davon aus, dass die Sonne scheint, aber kein Wind weht. Wenn man aber eine Aussage über die Qualität des Wetters machen will, hängt das nicht nur von einer, sondern von beiden Variablen ab. Man kann beispielsweise definieren, dass Sonnenschein und Windstill *gutes Wetter*, kein Sonnenschein und windig *schlechtes Wetter* und alles andere *mittelgutes Wetter* bedeutet.

Es gibt also insgesamt vier verschiedene Möglichkeiten, wie die Werte der Variablen verknüpft werden können:

SonneScheint	***WindWeht***	***Wetter***
true	true	gutes Wetter
true	false	mittelgutes Wetter
false	true	mittelgutes Wetter
false	false	schlechtes Wetter

Tabelle 4: Wetter

Diese Verknüpfung (*wenn die Sonne scheint und kein Wind weht*) kann man in C# mittels logischer Operatoren ausführen. Der logische Operator für die UND-Verknüpfung heißt `&&`. Das Programm sieht dann also folgendermaßen aus (s. Folgeseite):

```
public class Wetter
{
  public static void Main()
  {
    bool SonneScheint = true;
    bool WindWeht = false;

    bool MittelgutesWetter = SonneScheint && WindWeht;
  }
}
```

Code 7: Wetter 02

Je nachdem, wie die beiden Variablen *SonneScheint* und *WindWeht* nun gewählt werden, enthält die Variable *MittelgutesWetter* entweder den Wert true oder den Wert false. Beachten Sie, dass hiermit nur eine Überprüfung stattfindet, ob mittelgutes Wetter ist (und selbst hier wird nur einer von zwei Fällen abgefragt). Sollte die Variable *MittelgutesWetter* den Wert false enthalten, so kann sowohl schlechtes als auch gutes Wetter sein (oder sogar der andere Fall von mittelgutem Wetter!).

Um auch zwischen schlechtem und gutem Wetter differenzieren zu können, benötigt man eine weitere Variable, beispielsweise *SchlechtesWetter*. Es gibt in C# aber noch andere logische Operatoren außer der UND-Verknüpfung.

logischer Operator	***Funktion***	***Beschreibung***
&&	AND	A und B sind wahr
||	OR (inklusives Oder)	A oder B oder beide sind wahr
^	XOR (exklusives Oder)	entweder ist A wahr oder B
!	NOT	A ist nicht wahr

Tabelle 5: logische Operatoren

Welcher logische Operator bei welchen Eingangsdaten welches Ergebnis liefert, lässt sich am besten in einer Tabelle visualisieren.

&& (AND)		
	true	false
true	true	false
false	false	false
|| (OR)		
	true	false
true	true	true
false	true	false
^ (XOR)		
	true	false
true	false	true
false	true	false
! (NOT)		
true	false	
false	true	

Tabelle 6: Zustände logischer Operatoren

C# benutzt zur Auswertung logischer Operatoren ein System namens *Short cut evaluation*. Das bedeutet, dass die Auswertung abgebrochen wird, sobald das Ergebnis feststeht. Sind beispielsweise zwei Ausdrücke UND-verknüpft, aber der erste enthält den Wert false, so wird die Auswertung abgebrochen, da es egal ist, ob der den Wert zweite true oder false enthält.

7.5 Relationale Operatoren

Neben den arithmetischen und den logischen Operatoren gibt es noch einen weiteren wichtigen Typ: Die relationalen Operatoren. Sie dienen dazu, eine Relation zwischen zwei Ausdrücken zu beschreiben, also, um einen Vergleich anzustellen. Das wichtigste Einsatzgebiet relationaler Operatoren werden Sie im nächsten Kapitel kennen lernen.

In C# können Sie zwei Variablen auf verschiedene Arten vergleichen: Der einfachste Vergleich ist der Test auf Gleichheit. Dieser wird mit dem Operator `==` ausgeführt. Beachten Sie, dass hier zwei Gleichheitszeichen hintereinander geschrieben werden, während es bei der Zuweisung nur eines ist.

In anderen Programmiersprachen als C# entstehen dadurch oft Fehler, dass der Programmierer aus Versehen nur ein Gleichheitszeichen setzt, wo eigentlich zwei hingehören (oder umgekehrt). Diesen Fehler bemerkt der Compiler von C# jedoch, so dass Sie sich bei einem Fehler in Ihrem Programm hierüber keine Gedanken machen müssen.

Während der Test auf Gleichheit mittels `==` durchgeführt wird, wird die Ungleichheit mittels `!=` überprüft.

```
public class RelationaleOperatoren
{
  public static void Main()
  {
    int ErsteZahl = 3;
    int ZweiteZahl = 5;

    bool VariablenSindGleich;
    bool VariablenSindUngleich;

    VariablenSindGleich = (ErsteZahl == ZweiteZahl);
    VariablenSindUngleich = (ErsteZahl != ZweiteZahl);
  }
}
```

Code 8: Relationale Operatoren

Da die beiden Variablen *ErsteZahl* und *ZweiteZahl* nicht gleich sind, wird der Ausdruck

```
ErsteZahl == ZweiteZahl
```

den Wert `false` zurückliefern und ihn dann in *VariablenSindGleich* speichern.

Dagegen enthält *VariablenSindUngleich* den Wert true, da der Ausdruck

```
ErsteZahl != ZweiteZahl
```

wahr ist, da die beiden Zahlen unterschiedlich sind.

Außer den Operatoren für Gleichheit und Ungleichheit gibt es noch einige weitere, die im nächsten Kapitel (ebenso wie die logischen Operatoren) angewendet werden.

Operator	***Funktion***
==	gleich
!=	ungleich
<	kleiner
>	größer
<=	kleiner oder gleich
>=	größer oder gleich

Tabelle 7: Relationale Operatoren

7.6 Ausblick

Im nächsten Kapitel werden Sie lernen, Entscheidungen zu programmieren und dabei die logischen und relationalen Operatoren einzusetzen.

8 Entscheidungen

In diesem Kapitel werden Sie lernen, Entscheidungen zu programmieren und logische und relationale Operatoren in der Praxis einzusetzen.

8.1 Einleitung

Alle Programme, die Sie bisher geschrieben haben, waren sehr einfach aufgebaut. Ihnen allen ist gemeinsam, dass sie vom Computer von der ersten bis zur letzten Zeile linear abgearbeitet werden. Doch dieser Fall kommt in der Praxis sehr selten vor.
Sehr oft hängt der weitere Ablauf eines Programms beispielsweise von einer Eingabe des Benutzer ab. Je nachdem, was eingegeben wird, entscheidet das Programm, einige Codezeilen auszulassen und an einer anderen Stelle fortzusetzen. Für Entscheidungen gibt es in C# zwei Befehle: `if` und `switch`. Zunächst werden Sie die `if`-Anweisung kennen lernen.

8.2 Die einfache if-Anweisung

Mit if können Sie einen Ausdruck abfragen und abhängig davon, ob der Ausdruck wahr ist oder nicht, Anweisungen ausführen oder diese Anweisungen überspringen.

```
using System;

public class Entscheidung
{
  public static void Main()
  {
    string Name;

    Console.Write("Bitte geben Sie Ihren Namen ein: ");
    Name = Console.ReadLine();

    if(Name == "Bill Gates")
    {
      Console.WriteLine("Das ist ja unglaublich!");
    }

    Console.WriteLine("Hallo, " + Name);
  }
}
```

Code 9: Einfache if-Anweisung

Dieses Programm fordert Sie zunächst auf, Ihren Namen einzugeben. Dann überprüft es mittels des relationalen Operators `==` in der Zeile

```
if(Name == "Bill Gates")
```

ob der eingegebene Name *Bill Gates* ist. Wenn das der Fall ist, ist der Ausdruck

```
Name == "Bill Gates"
```

wahr und die Bedingung für die `if`-Anweisung ist erfüllt. Wird ein anderer Name eingegeben, so ist der Ausdruck falsch und die Bedingung der `if`-Anweisung nicht erfüllt. Wie Sie sehen, steht die Bedingung immer in runden Klammern hinter der `if`-Anweisung. Beachten Sie bitte, dass diese Zeile nicht durch ein Semikolon abgeschlossen wird.

Wenn die Bedingung für die `if`-Anweisung erfüllt ist, wird der Code ausgeführt, der in den geschweiften Klammern folgt. Das ist der sogenannte `if`-Block. In dem Beispiel besteht der `if`-Block also nur aus einer einzigen Zeile, nämlich

```
Console.WriteLine("Das ist ja unglaublich!");
```

Sie wird ausgeführt, sofern als Name *Bill Gates* eingegeben wurde. Die Zeile

```
Console.WriteLine("Hallo, " + Name);
```

steht hingegen nicht mehr innerhalb des `if`-Blocks und wird immer ausgeführt, egal welcher Name eingegeben wird. Die geschweiften Klammern müssen gesetzt werden, außer der `if`-Block besteht nur aus einer einzigen Zeile.

```
if(Name == "Bill Gates")
{
  Console.WriteLine("Das ist ja unglaublich!");
}
```

und

```
if(Name == "Bill Gates")
  Console.WriteLine("Das ist ja unglaublich!");
```

sind also gleichbedeutend. Wenn Sie sich allerdings angewöhnen, die geschweiften Klammern immer zu setzen, erhöht dies zum einen die Lesbarkeit Ihres Programms und zum anderen kann es Ihnen bei einer nachträglichen Erweiterung des `if`-Blocks nicht passieren, dass Sie vergessen, die Klammern doch noch zu setzen.

Die Syntax der einfachen `if`-Anweisung lautet also

```
if(Bedingung)
{
  Anweisungsblock;
}
```

8.3 Die erweiterte if-Anweisung

Es gibt aber noch eine erweiterte `if`-Anweisung, die dann benutzt wird, wenn Sie noch einen alternativen Anweisungsblock benötigen. Sie könnten solche Alternativen folgendermaßen programmieren (s. Folgeseite):

```
using System;

public class Entscheidung
{
  public static void Main()
  {
    string Name;

    Console.Write("Bitte geben Sie Ihren Namen ein: ");
    Name = Console.ReadLine();

    if(Name == "Bill Gates")
    {
      Console.WriteLine("Das ist ja unglaublich!");
    }
    if(Name != "Bill Gates")
    {
      Console.WriteLine("Sie arbeiten wohl nicht bei
      Microsoft!");
    }

    Console.WriteLine("Hallo, " + Name);
  }
}
```

Code 10: Erweiterte if-Anweisung

Es soll also ein bestimmter Anweisungsblock ausgeführt werden, wenn die Bedingung erfüllt ist, aber ein anderer, wenn die Bedingung nicht erfüllt ist. Da die Syntax mit zwei `if`-Anweisungen dazu aber zu umständlich ist, gibt es eine Vereinfachung.

Aus

```
if(Name == "Bill Gates")
{
  Console.WriteLine("Das ist ja unglaublich!");
}
if(Name != "Bill Gates")
{
  Console.WriteLine("Sie arbeiten wohl nicht bei
  Microsoft!");
}
```

wird dann

```
if(Name == "Bill Gates")
{
  Console.WriteLine("Das ist ja unglaublich!");
}
else
{
  Console.WriteLine("Sie arbeiten wohl nicht bei
  Microsoft!");
}
```

Der erste Anweisungsblock wird nur dann ausgeführt, wenn die Bedingung erfüllt ist, der zweite, der dem Schlüsselwort `else` folgt, nur dann, wenn die Bedingung nicht erfüllt ist. Beachten Sie bitte, dass auch nach `else` kein Semikolon folgt.

Die Syntax der erweiterten `if`-Anweisung lautet also

```
if(Bedingung)
{
  Anweisungsblock;
}
else
{
  Anweisungsblock;
}
```

Die Bedingung sowohl bei der einfachen als auch bei der erweiterten `if`-Anweisung muss irgendein Ausdruck sein, der als Wert true oder false ergibt, also beispielsweise ein Vergleich mittels relationaler Operatoren.

8.4 Verschachtelte if-Anweisungen

Stellen Sie sich vor, Sie möchten einen Anweisungsblock nur dann ausführen, wenn nicht eine, sondern sogar zwei Bedingungen erfüllt sind. Dazu können Sie `if`-Anweisungen verschachteln, wobei Sie die einfache und die erweiterte Form willkürlich mischen können.

```
using System;

public class Entscheidung
{
  public static void Main()
  {
    string Name;
    string Passwort;

    Console.Write("Bitte geben Sie Ihren Namen ein: ");
    Name = Console.ReadLine();

    if(Name == "Bill Gates")
    {
      Console.Write("Bitte geben Sie Ihr Passwort ein: ");
      Passwort = Console.ReadLine();

      if(Passwort == "Microsoft")
      {
        Console.WriteLine("Sie müssen wirklich Bill Gates
        sein!");
      }
      else
      {
        Console.WriteLine("Sie geben sich wohl gern für
        jemand anders aus!");
      }
    }

    Console.WriteLine("Hallo, " + Name);
  }
}
```

Code 11: Verschachtelte if-Anweisung

Das Programm fragt neben dem Namen auch noch nach einem Passwort. Angenommen, der Benutzer gibt nicht *Bill Gates* ein, dann wird er vom Programm einfach nur mit einem simplen *Hallo* begrüßt. Gibt er aber *Bill Gates* ein, so möchte das Programm auch noch ein Passwort wissen. Ist das Passwort richtig, so wird der Benutzer als Bill Gates anerkannt, ist es aber falsch, so hat er sich entweder vertippt oder er ist nicht Bill Gates.

Wenn Sie noch mehr als zwei Bedingungen testen wollen, werden geschachtelte `if`-Anweisungen schnell unübersichtlich. Dieses Problem können Sie mit den logischen Operatoren, die Sie im letzten Kapitel kennengelernt haben, umgehen.

Die beiden `if`-Anweisungen

```
if(Name == "Bill Gates")
{
  if(Passwort == "Microsoft")
  {
    Console.WriteLine("Sie müssen wirklich Bill Gates
    sein!");
  }
}
```

können Sie nämlich zu einer zusammenfassen, denn die Zeile

```
Console.WriteLine("Sie müssen wirklich Bill Gates
sein!");
```

soll nur ausgeführt werden, wenn der Name und das Passwort richtig sind. Also schreiben Sie statt der verschachtelten `if`-Anweisungen

```
if(Name == "Bill Gates" && Passwort == "Microsoft")
```

und fragen so beide Bedingungen in einer Zeile ab. Um die Reihenfolge der Auswertung etwas deutlicher zu machen, können Sie noch zusätzliche Klammern setzen. Diese sind allerdings nicht notwendig, sondern dienen nur der Übersicht.

```
if((Name == "Bill Gates") && (Passwort == "Microsoft"))
```

8.5 Die else if-Anweisung

Wenn Sie sich zwischen mehr als zwei Alternativen entscheiden können, werden verschachtelte `if`-Anweisungen schnell unübersichtlich, da die Einrückebene ständig weiter nach rechts rutscht. Deshalb können Sie in solchen Fällen die `else if`-Anweisung verwenden.

```
using System;

public class Entscheidung
{
  public static void Main()
  {
    string Name;

    Console.Write("Bitte geben Sie Ihren Namen ein: ");
    Name = Console.ReadLine();

    if(Name == "Bill Gates")
    {
      Console.WriteLine("Sie arbeiten wohl bei
      Microsoft!");
    }
    else if(Name == "Steve Jobs")
    {
      Console.WriteLine("Sie lieben Ihren Mac sicherlich
      über alles!");
    }
    else if(Name == "Steve Case")
    {
      Console.WriteLine("Wetten, dass Sie über AOL ins
      Internet kommen!");
    }
    else
    {
      Console.WriteLine("Tut mir leid, aber Sie kenne ich
      nicht!");
    }
  }
}
```

Code 12: Die else if-Anweisung

8.6 Der Bedingungsoperator

Neben verkürzenden Schreibweisen bei Rechen- und Zuweisungsoperatoren unterstützt C# auch eine verkürzende Schreibweise der erweiterten `if`-Anweisung. Dazu wird der sogenannte Bedingungsoperator `?:` verwendet. Er ist der einzige Operator von C#, der drei Argumente erwartet.

Seine Syntax lautet

```
Bedingung ? Anweisung : Anweisung
```

, wobei die erste Anweisung ausgeführt wird, falls die Bedingung wahr ist, die zweite jedoch, falls die Bedingung falsch ist.

Statt

```
if(i % 2 == 0)
  ZahlGerade = true;
else
  ZahlGerade = false;
```

kann man mittels des Bedingungsoperators auch

```
ZahlGerade = (i % 2 == 0) ? true : false;
```

schreiben. Der Variablen *ZahlGerade* wird dabei der Wert `true` zugewiesen, falls die Bedingung `i % 2 == 0` wahr ist, andernfalls `false`.

Der Bedingungsoperator kann manchmal nützlich sein, um einfache `if`-Abfragen, bei denen beide Anweisungsblöcke nur aus einer Anweisung oder einem Ausdruck bestehen, zu verkürzen. Da er aber bei längeren Konstruktionen äußerst schnell unübersichtlich wird, sollte man ihn zurückhaltend einsetzen.

8.7 Die switch-Anweisung

Wenn Sie sich zwischen mehr als zwei Alternativen entscheiden müssen, können Sie alternativ auch die `switch`-Anweisung benutzen. Im Kopf der `switch`-Anweisung wird ein Ausdruck übergeben, der ausgewertet werden soll, und im Körper können Sie dann alle wichtigen Werte abfragen.

```
using System;

public class Entscheidung
{
  public static void Main()
  {
    string Name;

    Console.Write("Bitte geben Sie Ihren Namen ein: ");
    Name = Console.ReadLine();

    switch(Name)
    {
      case "Bill Gates":
        Console.WriteLine("Sie arbeiten wohl bei
         Microsoft!");
        break;
      case "Steve Jobs":
        Console.WriteLine("Sie lieben Ihren Mac
         sicherlich über alles!");
        break;
      case "Steve Case":
        Console.WriteLine("Wetten, dass Sie über AOL ins
         Internet kommen!");
        break;
      default:
        Console.WriteLine("Tut mir leid, aber Sie kenne
         ich nicht!");
        break;
    }
  }
}
```

Code 13: Die switch-Anweisung

In diesem Beispiel ist der Ausdruck, der ausgewertet werden soll, einfach die Variable *Name*. Anders als bei der `if`-Anweisung ist der Typ der auszuwertenden Variable allerdings nicht beliebig. Mittels `switch` lassen sich nämlich nur ganzzahlige Datentypen und Zeichenketten auswerten, so dass man unter Umständen doch auf die `if`-Anweisung zurückgreifen muss.

```
switch(Name)
```

Beachten Sie bitte, dass wie bei der `if`-Anweisung auch hinter `switch` kein Semikolon steht. Im Körper der `switch`-Anweisung, der durch geschweifte Klammern eingeschlossen wird, werden die verschiedenen Möglichkeiten mittels `case` abgefragt.

```
case "Bill Gates":
```

Diese Zeile vergleicht die Variable *Name* mit dem Ausdruck *Bill Gates*, und falls sie identisch sind, wird der entsprechende Zweig ausgeführt und der Text *Sie arbeiten wohl bei Microsoft!* auf dem Bildschirm ausgegeben.

Wenn kein Name eingegeben wird, der einem der drei in den `case`-Zweigen entspricht, wird der `default`-Zweig ausgeführt (der `default`-Zweig entspricht also dem `else` bei der `if`-Anweisung). Er ist aber wie der `else`-Zweig optional, das heißt, Sie können ihn auch weglassen.

Jeder Zweig der `switch`-Anweisung (auch der `default`-Zweig) muss durch das Schlüsselwort `break` abgeschlossen werden. Wenn Sie ein `break` weglassen, meldet der Compiler einen Fehler. Die Syntax der `switch`-Anweisung lautet also

```
switch(Ausdruck)
{
  case VergleichsAusdruck1:
    Anweisungsblock;
    break;
  case VergleichsAusdruck2:
    Anweisungsblock;
    break;
  ...
  case VergleichsAusdruckN:
    Anweisungsblock;
    break;
  default:
    Anweisungsblock;
    break;
}
```

8.8 Ausblick

Im nächsten Kapitel werden Sie lernen, was Schleifen sind und welche Arten von ihnen es in C# gibt.

9 Schleifen

In diesem Kapitel werden Sie lernen, was Schleifen sind und welche Arten von ihnen es in C# gibt.

9.1 Einleitung

Nachdem Sie im letzten Kapitel bereits gelernt haben, wie Sie den linearen Ablauf von Programmen mit Entscheidungen durchbrechen können, lernen Sie nun noch eine weitere Technik dazu kennen.

Schleifen dienen dazu, Anweisungen wiederholt auszuführen, wobei entweder eine bestimmte Anzahl vorgegeben wird oder der Ablauf dynamisch gesteuert werden kann.

9.2 Die for-Schleife

Die einfachste Schleife in C# ist die `for`-Schleife. Sie ist eine einfache Zählschleife und führt die Anweisungen in ihrem Körper eine bestimmte Anzahl mal durch. Die Syntax lässt sich am Besten an einem Beispiel verdeutlichen.

```
using System

public class Schleife
{
  public static void Main()
  {
    int i;

    for(i = 0; i < 10; i++)
    {
      Console.WriteLine("Hallo!");
    }
  }
}
```

Code 14: Die for-Schleife

Dieses Programm gibt den Text *Hallo!* zehnmal untereinander auf den Bildschirm aus. Die ganze Steuerung der Schleife steckt in der einen Zeile

```
for(i = 0; i < 10; i++)
```

Hier wird zunächst mittels

```
i = 0
```

der Initialisierungsausdruck der Schleife gesetzt, das heißt, die Variable *i* wird auf 0 gesetzt. Dies findet nur ein einziges Mal, nämlich vor Beginn der Schleife, statt. Mittels

```
i < 10
```

wird die Abbruchbedingung gesetzt, das heißt, die Schleife soll ausgeführt werden, so lange diese Bedingung wahr ist (also *i* kleiner als 10 ist). Damit sich *i* aber überhaupt ändert, wird mittels

```
i++
```

der Aktualisierungsausdruck festgelegt, dass also *i* nach jedem Schleifendurchlauf um eines erhöht wird. Ist *i* dann immer noch kleiner als 10, so wird ein neuer Schleifendurchlauf gestartet.

Beachten Sie bitte, dass die Zeile, in der die `for`-Anweisung steht, nicht durch ein Semikolon abgeschlossen wird. Der Anweisungsblock muss übrigens wie bei der `if`-Anweisung nicht in geschweiften Klammern stehen, wenn er nur aus einer Zeile besteht, es empfiehlt sich aber, die Klammern trotzdem immer zu setzen.

Die Syntax der `for`-Schleife lautet also

```
for(Initialisierungsausdruck; Abbruchbedingung;
Aktualisierungsausdruck)
{
  Anweisungsblock;
}
```

Die Variable *i* heißt Schleifenvariable und kann innerhalb der Schleife auch verwendet werden. Folgendes Programm berechnet beispielsweise die Quadratzahlen von 1 bis 10.

```
using System;

public class Quadratzahlen
{
  public static void Main()
  {
    int i;
    for(i = 1; i <= 10; i++)
    {
      Console.WriteLine("Das Quadrat von " + i + " ist "
       + (i * i) + "!");
    }
  }
}
```

Code 15: Quadratzahlen von 1 bis 10

Es hat sich in der Praxis eingebürgert, Schleifenvariablen mit Kleinbuchstaben (bei *i* beginnend) zu bezeichnen. Sie können die Variablen aber auch anders benennen. Außerdem ist es gebräuchlich, die Schleifenvariable mit 0 und nicht mit 1 zu initialisieren, solange es in der Schleife nicht auf eventuell andere Werte ankommt (wie in dem Beispiel, wo die Quadratzahlen ab 1 und nicht ab 0 berechnet werden sollen).

Es gibt übrigens noch die Möglichkeit, die Variable *i* gleich mit der Schleife zu deklarieren, so dass Sie sich eine Zeile sparen können.

```
for(int i = 1; i <= 10; i++)
```

Allerdings ist diese Schreibweise auch fehleranfälliger, da man nicht auf Anhieb sieht, an welcher Stelle des Programms eine Variable deklariert wird.

Es gibt übrigens noch eine Ergänzung zum Befehl `System.Console.WriteLine`. Wenn Sie viele Strings aneinanderhängen, wird die Syntax mit den vielen Stringkonkatenationsoperatoren unübersichtlich. Deshalb können Sie statt

```
Console.WriteLine("Das Quadrat von " + i + " ist " + (i *
i) + "!");
```

auch

```
Console.WriteLine("Das Quadrat von {0} ist {1}!", i, i *
i);
```

schreiben. Hier geben Sie also zunächst den konstanten Text an und schreiben an die Stellen, wo die Ausdrücke eingesetzt werden sollen, in geschweifte Klammern eingeschlossene Zahlen (bei 0 beginnend). Hinter diesem String schließen Sie dann die durch Kommata getrennten einzelnen Ausdrücke an.

9.3 Die break- und die continue-Anweisung

Manchmal kann es passieren, dass der Benutzer einen vorzeitigen Abbruch einer Schleife veranlassen möchte. Beispielsweise könnte eine `for`-Schleife hundert mal durchlaufen werden, aber in jedem Durchlauf wird der Benutzer gefragt, ob er abbrechen möchte. Ein solcher Abbruch erfolgt mittels der `break`-Anweisung, die Sie schon bei der `case`-Anweisung kennengelernt haben.

```
using System;

public class Quadratzahlen
{
  public static void Main()
  {
    int i;
    for(i = 1; i <= 100; i++)
    {
      Console.WriteLine("Das Quadrat von {0} ist {1}!",
       i, i * i);
      Console.Write("Möchten Sie die nächste Quadratzahl
       berechnen (j / n)? ");
      if(Console.ReadLine() == "n")
      {
        break;
      }
    }
    Console.WriteLine("So, fertig!");
  }
}
```

Code 16: Die break-Anweisung

Gibt der Benutzer in irgendeinem Durchlauf der Schleife ein *n* ein, so wird die Anweisung

```
break;
```

ausgeführt und aus der Schleife herausgesprungen. Als nächste (und letzte) Anweisung wird dann nur noch der Text *So, fertig!* ausgegeben. Bei mehreren verschachtelten Schleifen springt `break` aus der inneren in die äußere Schleife und setzt dort die Ausführung fort.

Eine weitere Anweisung zur Schleifensteuerung ist `continue`. Mit der `continue`-Anweisung kann man den aktuellen Schleifendurchlauf abbrechen und den nächsten starten.

```
using System;

public class Quadratzahlen
{
  public static void Main()
  {
    int i;
    for(i = 1; i <= 100; i++)
    {
      Console.Write("Möchten Sie das Quadrat von {0}
       ausgeben (j / n)?", i);
      if(Console.ReadLine() != "j")
      {
        continue;
      }
      Console.WriteLine("Das Quadrat von {0} ist {1}!",
       i, i * i);
    }
  }
}
```

Code 17: Die continue-Anweisung

Antwortet der Benutzer auf die Frage, ob er das Quadrat der aktuellen Zahl ausgeben möchte, nicht mit einem *j*, wird die `continue`-Anweisung ausgeführt und der nächste Schleifendurchlauf gestartet, das heißt, die letzte Anweisung wird übersprungen.

9.4 Die while-Schleife

Der Nachteil der `for`-Schleife ist, dass von vornherein feststehen muss, wie oft sie durchlaufen werden soll. Eine Schleife, die so lange läuft, bis (irgendwann) ein bestimmtes Ereignis eintritt, ist so nicht zu realisieren. Für solche Aufgaben gibt es in C# die `while`-Schleife.

Im Kopf der `while`-Schleife wird ein Ausdruck übergeben, und so lange dieser wahr ist, wird die Schleife ausgeführt. Sie müssen aber darauf achten, dass die Bedingung auf jeden Fall irgendwann wahr werden kann, sonst befindet sich das Programm in einer Endlosschleife.

```
using System;

public class Kreis
{
  public static void Main()
  {
    const double Pi = Math.PI;
    double Flaeche;
    int Radius = 1;
    bool NochmalBerechnen = true;

    while(NochmalBerechnen)
    {
      Flaeche = Pi * Radius * Radius;
      Console.Write("Der Kreis mit dem Radius {0} hat ",
      Radius);
      Console.WriteLine("die Fläche {0}!", Flaeche);
      Radius++;
      Console.Write("Möchten Sie die Fläche des ");
      Console.Write("nächsten Kreises berechnen (j / n)?
      ");
      if(Console.ReadLine() != "j")
      {
        NochmalBerechnen = false;
      }
    }
  }
}
```

Code 18: Die while-Schleife

Dieses Programm berechnet nacheinander die Flächen von Kreisen mit dem Radius 1, 2, 3, ... so lange, bis der Anwender eingibt, dass er keine weitere Fläche mehr berechnen möchte, also die Variable *NochmalBerechnen* auf `false` gesetzt wird. Beachten Sie bitte, dass hinter der `while`-Anweisung kein Semikolon kommt. Die geschweiften Klammern sind optional, solange der Schleifenkörper aus nur einer Zeile besteht.

Die Syntax der `while`-Schleife lautet also

```
while(Bedingung)
{
  Anweisungsblock;
}
```

In einem Punkt verhält sich die `while`- genauso wie die `for`-Schleife, sie sind nämlich beide abweisend. Das heißt, dass die Schleife unter Umständen nicht ein einziges Mal durchlaufen wird. Das kann passieren, wenn die Bedingung von vornherein falsch ist.

```
bool Bedingung = false;
while(Bedingung)
{
   ...
}
```

wird beispielsweise nie ausgeführt, da die Variable *Bedingung* von Anfang an den Wert `false` enthält. Genauso wenig wird die `for`-Schleife

```
for(i = 1; i > 100; i++)
{
   ...
}
```

ausgeführt, da *i* am Anfang den Wert 1 hat, die Schleife aber ausgeführt werden soll, solange *i* größer als 100 ist.

Vielleicht ist Ihnen in dem Programm außerdem die Zeile

```
const double Pi = Math.PI;
```

aufgefallen. Hier wird eine Variable *Pi* vom Typ `double` definiert, aber zusätzlich mit dem Schlüsselwort `const` versehen. Dadurch kann der Wert von *Pi* im Lauf des Programms nicht mehr verändert werden, sondern *Pi* stellt eine Konstante dar.

Der Wert einer Konstanten muss ihr gleich bei der Deklaration übergeben werden, da Sie später nicht mehr schreibend auf die Konstante zugreifen können. Als Wert wird in diesem Fall `Math.PI` übergeben.

PI ist eine vom .net Framework vorgegebene Konstante, die in Math enthalten ist (so wie WriteLine in Console enthalten ist), Math wiederum ist System untergeordnet. Math enthält mathematische Funktionen und wichtige mathematische Konstanten wie Pi oder die Zahl e. Natürlich hätten Sie in dem Programm auch direkt mit System.Math.PI arbeiten können, statt diesen Wert der Konstante *Pi* zuzuweisen.

9.5 Die do-Schleife

Es gibt noch eine weitere Schleife in C#, die der while-Schleife sehr ähnlich ist, nämlich die do-Schleife. Sie unterscheidet sich von der while-Schleife nur in dem Punkt, dass sie nicht abweisend ist und somit auf jeden Fall mindestens ein Schleifendurchlauf stattfindet.

```
using System;

public class Kreis
{
  public static void Main()
  {
    const double Pi = Math.PI;
    double Flaeche;
    int Radius = 1;
    bool NochmalBerechnen = true;

    do
    {
      Flaeche = Pi * Radius * Radius;
      Console.Write("Der Kreis mit dem Radius {0} hat ",
       Radius);
      Console.WriteLine("die Fläche {0}!", Flaeche);
      Radius++;
      Console.Write("Möchten Sie die Fläche des ");
      Console.Write("nächsten Kreises berechnen (j / n)?
      ");
      if(Console.ReadLine() != "j")
      {
        NochmalBerechnen = false;
      }
    }
    while(NochmalBerechnen);
  }
}
```

Code 19: Die do-Schleife

Dieses Programm hat genau die selbe Wirkung wie das Beispiel der `while`-Schleife, aber es ist mit einer `do`-Schleife programmiert. Die Syntax der `do`-Schleife lautet also

```
do
{
  Anweisungsblock;
}
while(Bedingung);
```

Beachten Sie bitte, dass hinter dem `do` kein Semikolon steht, hinter dem `while` aber sehr wohl eines! Stünde hier keins, könnte der Compiler das Ende der `do`-Schleife und den Anfang einer `while`-Schleife nicht unterscheiden.

Auch wenn es nicht noch einmal ausdrücklich erwähnt wurde, funktionieren die `break`- und die `continue`-Anweisung in der `while`- und der `do`-Schleife auf die selbe Art und Weise wie bei der `for`-Schleife.

9.6 Ausblick

Im nächsten Kapitel werden Sie Arrays kennen lernen und erfahren, wie man Strings miteinander vergleichen kann.

10 Arrays

In diesem Kapitel werden Sie lernen, was Arrays sind und wie Sie Strings miteinander vergleichen können.

10.1 Einleitung

Nachdem Sie im letzten Kapitel Schleifen kennengelernt haben, werden Sie in diesem Kapitel eine etwas sinnvollere Anwendung als bisher entwickeln. Es geht dabei um ein Programm, welches Daten vom Benutzer einliest und diese dann manipulieren und wieder ausgeben kann.

Für den Anfang soll unser Programm Namen einlesen und diese alphabetisch sortiert wieder ausgeben. Das Problem in dieser Anwendung ist die Speicherung der Daten, denn selbstverständlich können Sie für drei oder vier Namen noch Variablen anlegen, aber es wäre viel zu viel Aufwand, hundert oder mehr Variablen händisch anzulegen.

10.2 Arrays

Um dieses Problem zu lösen, gibt es in C# eine Datenstruktur namens *Array*. Beachten Sie, dass es sich hierbei um keinen Datentyp handelt, sondern um eine Datenstruktur!

Ein Array können Sie sich wie eine Liste vorstellen, in der nacheinander alle Werte gespeichert werden. Auf einen einzelnen Wert greifen Sie zu, indem Sie den Namen der Liste und einen Index verwenden, der angibt, an welcher Position in der Liste sich der Wert enthält.

Arrays enthalten dabei allerdings nur Werte von ein und demselben Datentyp, Sie können also nicht Zahlen und Text innerhalb eines Array mischen (es gibt eine Ausnahme, nämlich den Datentyp `object`, den Sie später noch kennen lernen werden).

Ein Array legen Sie mittels des `new`-Operators an. Um ein Array von beispielsweise zehn Integer-Werten zu erzeugen, geben Sie folgende Zeile ein:

```
new int[10];
```

Allerdings können Sie dieses Array noch nicht verwenden, denn es hat keinen Namen. Um also ein "sinnvolles" Array anzulegen, muss die Zeile

```
int[] IntegerArray = new int[10];
```

lauten. Hiermit haben Sie ein Array für Integerwerte mit zehn Elementen angelegt, welches *IntegerArray* heißt. Auf die einzelnen Elemente können Sie jetzt mittels der Indizes 0 bis 9 zugreifen, also auf das erste Element mit

```
IntegerArray[0]
```

und auf das letzte mit

```
IntegerArray[9]
```

Beachten Sie, dass die Zählung der Indizes bei 0 startet und nicht bei 1! Der höchste Index eines Arrays mit n Elementen ist also immer n - 1!

Um einem Element eines Arrays einen Wert zuzuweisen, gehen Sie genauso vor, als ob Sie einer normalen Variablen einen Wert zuweisen wollten, indem Sie den Zuweisungsoperator benutzen.

```
IntegerArray[2] = 27;
```

weist beispielsweise dem dritten Element den Wert 27 zu. Sie können die Werte aber auch gleich bei der Deklaration des Arrays mit angeben, dann übernimmt C# für Sie sogar die Aufgabe, die Anzahl der Elemente zu ermitteln.

Statt

```
int[] IntegerArray = new int[10];
IntegerArray[0] = 27;
IntegerArray[1] = 42;
...
IntegerArray[8] = 7;
IntegerArray[9] = 1083;
```

können Sie also auch

```
int[] IntegerArray = {27, 42, ..., 7, 1083};
```

schreiben. Sie können aber auch Arrays anlegen, bei denen die Größe erst zur Laufzeit des Programms feststeht, indem Sie für die Größenangabe einer Variable verwenden.

```
int i = 5;
double[] DoubleArray = new double[i];
```

legt beispielsweise ein Array für Double-Werte der Größe fünf an.

10.3 Ein- und Ausgabe der Daten

Nun können Sie bereits den ersten Teil des Programms realisieren, nämlich die Dateneingabe. Dazu muss das Programm ein `string`-Array anlegen und in einer Schleife die Namen vom Benutzer abfragen.

```
using System;

public class NamenSortieren
{
  public static void Main()
  {
    int i;
    int AnzahlNamen = 10;
    string[] Name = new string[AnzahlNamen];

    for(i = 0; i < AnzahlNamen; i++)
    {
      Console.Write("Bitte geben Sie den {0}. Namen ein:
      ", i + 1);
      Name[i] = Console.ReadLine();
    }
  }
}
```

Code 20: Eingabe und Speicherung in einem Array

Wenn Sie dieses Programm kompilieren, wird es Sie der Reihe nach zehn Namen abfragen und diese nacheinander in dem Array *Name* speichern. Durch die Schleifenvariable *i* wird jedesmal in ein anderes Element des Arrays geschrieben.

Die Ausgabe ist ebenso einfach zu implementieren.

```
using System;

public class NamenSortieren
{
  public static void Main()
  {
    // Initialisierung der Variablen und des Arrays
    int i;
    int AnzahlNamen = 10;
    string[] Name = new string[AnzahlNamen];

    // Einlesen der Daten
    for(i = 0; i < AnzahlNamen; i++)
    {
      Console.Write("Bitte geben Sie den {0}. Namen ein:
      ", i + 1);
      Name[i] = Console.ReadLine();
    }

    // Ausgabe der Daten
    for(i = 0; i < AnzahlNamen; i++)
    {
      Console.WriteLine(Name[i]);
    }
  }
}
```

Code 21: Ausgabe von Daten aus einem Array

Dadurch, dass Sie die Arraygröße (und damit die Anzahl der Schleifedurchläufe) in der Variablen *AnzahlNamen* abgespeichert haben, brauchen Sie nur eine Zeile im Programm zu ändern, um mit einer anderen Anzahl von Namen zu arbeiten.

10.4 Sortieren

Nun fehlt nur noch der Sortieralgorithmus. In diesem Beispiel werden Sie einen sogenannten Bubble sort programmieren - auch wenn er nicht gerade zu den schnellsten Algorithmen gehört - , da er sehr einfach zu verstehen ist. Bubble sort arbeitet mit ständigen Vertauschungen von zwei benachbarten Elementen. Falls das erste größer ist als das zweite, werden sie vertauscht, ansonsten bleiben sie, wie sie sind. Dann macht er das selbe mit den nächsten beiden Elementen. Das ganze muss er so oft machen, bis in einem Durchlauf nichts mehr vertauscht wurde, denn dann ist das Array sortiert.

```
using System;

public class NamenSortieren
{
  public static void Main()
  {
    // Initialisierung der Variablen und des Arrays
    int i;
    int j;
    int AnzahlNamen = 10;
    bool ListeSortiert = false;
    string[] Name = new string[AnzahlNamen];
    string Temp;

    // Einlesen der Daten
    for(i = 0; i < AnzahlNamen; i++)
    {
      Console.Write("Bitte geben Sie den {0}. Namen ein:
      ", i + 1);
      Name[i] = Console.ReadLine();
    }

    // Sortieren der Daten mittels Bubble sort
    i = AnzahlNamen - 1;
    while((i >= 1) && (!ListeSortiert))
    {
      ListeSortiert = true;
      for(j = 0; j <= i - 1; j++)
      {
        if(String.Compare(Name[j], Name[j + 1]) > 0)
        {
          // Namen vertauschen
          Temp = Name[j];
          Name[j] = Name[j + 1];
          Name[j + 1] = Temp;
          ListeSortiert = false;
        }
      }
      i--;
    }

    // Ausgabe der Daten
    for(i = 0; i < AnzahlNamen; i++)
    {
      Console.WriteLine(Name[i]);
    }
  }
}
```

Code 22: Daten ins Array schreiben, sortieren und ausgeben

10.5 Referenzdatentypen

In dem Programm gibt es eine bemerkenswerte Zeile, nämlich die, in der die beiden Strings verglichen werden. Vielleicht hätten Sie erwartet, dass dies mittels eines relationalen Operators möglich ist. Wenn Sie es versuchen, erhalten Sie vom Compiler eine Fehlermeldung, dass der Operator `>` nicht auf den Typ `string` angewendet werden kann.

Auch wenn Sie davon nichts bemerkt haben, ist `string` nicht mit den meisten anderen Datentypen wie beispielsweise `int`, `double` oder `char` vergleichbar. Die Datentypen `int`, `double`, `char`, ... sind sogenannte *elementare* Datentypen, denn sie enthalten direkt einen Wert. Der Datentyp string hingegen ist ein *Referenzdatentyp*, denn er enthält nur einen Verweis auf eine Stelle im Speicher, in der der eigentliche String steht.

Wenn Sie also zwei Strings mittels `>` vergleichen würden, würden Sie nicht überprüfen, ob der erste String (alphabetisch) größer als der zweite ist, sondern, ob die Speicherstelle des ersten eine höhere Adresse als die des zweiten hat. Der Datentyp `string` ist aber nicht der einzige Referenztyp, Arrays gehören beispielsweise auch dazu.

Um Strings trotzdem vergleichen zu können, gibt es in der Klasse `System.String` die Methode `Compare`, der Sie die beiden zu vergleichenden Strings übergeben. Da Sie `System` mittels der Anweisung `using` in der ersten Zeile bereits bekannt gegeben haben, müssen Sie nur noch `String.Compare` schreiben. Die Methode liefert eine Zahl als Rückgabewert zurück, die angibt, welcher String größer ist.

Rückgabewert von String.Compare()	***Bedeutung***
negative Zahl	der erste String ist kleiner als der zweite
0	beide Strings sind identisch
positive Zahl	der erste String ist größer als der zweite

Tabelle 8: Rückgabewert von StringCompare()

10.6 Arrays mit mehr als einer Dimension

Bisher haben Sie Arrays als Listen betrachtet, also nur als Arrays mit einer Dimension. Sie können aber auch genauso Arrays mit mehr Dimensionen anlegen, sogar mit einer rein theoretisch beliebig hohen Zahl an Dimensionen.

Ein Array mit mehreren Dimensionen legen Sie genau so wie ein Array mit einer Dimension an, mit dem Unterschied, dass Sie nun mehrere, durch Kommate getrennte Dimensionsgrößen angeben müssen.

Ein Beispiel für den Einsatz eines Arrays mit zwei Dimensionen wäre beispielsweise eine Matrix, in der jede Zeile einer Person entspricht und in den Spalten die dazugehörigen Daten wie Name, Vorname, Geburtstag, ... gespeichert sind. Als Beispiel können Sie einfach das Programm zur Sortierung der Namen abändern, so dass es auch die Vornamen speichert (die beim Vertauschen dann natürlich auch vertauscht werden müssen).

```
using System;

public class NamenSortieren
{
  public static void Main()
  {
    // Initialisierung der Variablen und des Arrays
    int i;
    int j;
    int AnzahlNamen = 10;
    bool ListeSortiert = false;
    string[,] Person = new string[AnzahlNamen, 2];
    string TempNachname;
    string TempVorname;

    // Einlesen der Daten
    for(i = 0; i < AnzahlNamen; i++)
    {
      Console.Write("Bitte geben Sie den {0}. Nachnamen
       ein: ", i + 1);
      Person[i, 0] = Console.ReadLine();
      Console.Write("Bitte geben Sie den entsprechenden
       Vornamen ein: ", i + 1);
      Person[i, 1] = Console.ReadLine();
    }
    // Sortieren der Daten mittels Bubble sort
    i = AnzahlNamen - 1;
    while((i >= 1) && (!ListeSortiert))
    {
      ListeSortiert = true;
      for(j = 0; j <= i - 1; j++)
      {
        if(String.Compare(Person[j, 0], Person[j + 1, 0])
         > 0)
        {
          // Namen vertauschen
          TempNachname = Person[j, 0];
          TempVorname = Person[j, 1];
          Person[j, 0] = Person[j + 1, 0];
          Person[j, 1] = Person[j + 1, 1];
          Person[j + 1, 0] = TempNachname;
          Person[j + 1, 1] = TempVorname;
          ListeSortiert = false;
        }
      }
      i--;
    }
    // Ausgabe der Daten
    for(i = 0; i < AnzahlNamen; i++)
    {
      Console.WriteLine("{0}, {1}", Person[i, 0],
       Person[i, 1]);
    }
  }
}
```

Code 23: Mehrdimensionale Arrays

Beachten Sie, dass bei diesem Programm nicht nur der lesende und schreibende Zugriff auf die Elemente des Arrays über zwei Indizes abläuft, sondern auch der Datentyp des Arraynamens an die Dimensionen angepasst werden muss.

```
string[,] Person = new string[AnzahlNamen, 2];
```

Ohne das Komma meldet der Compiler, dass die Anzahl der bei Elementzugriffen verwendeten Indizes nicht mit der deklarierten Anzahl übereinstimmt. Die Anzahl der Dimensionen eines Arrays wird in C# übrigens Rang genannt.

10.7 Arraygröße ermitteln

Meistens ist bekannt, wie groß ein Array ist. Es kann jedoch vorkommen, dass die Arraygröße während der Laufzeit dynamisch gesetzt wird und beispielsweise von einer Eingabe des Benutzers oder einer Zufallszahl abhängt. Um dann nicht jedes Mal eine komplizierte Berechnung ausführen zu müssen, gibt es eine Funktion, die den höchsten Index (also die Anzahl der Arrayelemente weniger eins) zurückliefert:
`GetUpperBound`

In dem Programmsegment

```
string[] Name = new string[10];
int LetzterIndex = Name.GetUpperBound(0);
```

wird der Variablen *LetzterIndex* also beispielsweise der Wert 9 zugewiesen, da ein Array mit 10 Elementen als höchsten Index den Wert 9 enthält. In den Klammern hinter `GetUpperBound` müssen Sie die Dimension angeben, für die Sie den höchsten Index ermitteln wollen.

Außer `GetUpperBound` gibt es auch eine Funktion `GetLowerBound`, die den niedrigsten Index für die übergebene Dimension zurückliefert. Dieser muss nämlich nicht zwingend 0 sein. Da andere Indizes als 0 aber nicht den .net-Konventionen entsprechen, soll darauf hier nicht näher eingegangen werden.

Schließlich gibt es noch eine Funktion, um die Gesamtzahl aller Elemente (aus allen Dimensionen) eines Arrays zu ermitteln: `Length`. Bei eindimensionalen Arrays liefert `Length` also den höchsten Index plus eins zurück.

```
string[] Name = new string[10];
int AnzahlElemente = Name.Length;
```

In diesem Beispiel wird der Variablen *AnzahlElemente* der Wert 10 zugewiesen, da das Array 10 Elemente enthält (auch wenn diese alle leer sind). Außer der Gesamtanzahl der Elemente können Sie auch den Rang eines Arrays mittels `Rank` abfragen.

```
string[,,] Person = new string[10, 3, 12];
int Rang = Person.Rank;
```

Die Variable *Rang* enthält in diesem Beispiel den Wert 3.

10.8 Die foreach-Schleife

Wenn Sie der Reihe nach alle Elemente eines Array bearbeiten möchten, gibt es dafür einen eigenen Schleifentyp, so dass Sie sich gar nicht um die Größe des Arrays kümmern müssen. Es handelt sich dabei um die `foreach`-Schleife. Die `foreach`-Schleife durchläuft das ganze Array einmal und liefert in jedem Durchlauf das nächste Element zurück.

Beachten Sie bei der `foreach`-Schleife bitte, dass sie im Vergleich zu den anderen Schleifentypen sehr eingeschränkt ist, da ihr Ablauf nicht gesteuert werden kann.

Daher eignet sich die `foreach`-Schleife nur in solchen Fällen, in denen Sie alle Elemente eines Arrays in einer festgelegten Reihenfolge durchlaufen wollen, und in denen der Index des jeweiligen Elementes keine Rolle spielt, sondern nur der Wert.

Das folgende Programm legt ein Integerarray der Größe 99 an und schreibt die Zahlen von 2 bis 100 hinein. Dann wird mittels einer `foreach`-Schleife für jede Zahl berechnet, ob es sich dabei um eine Primzahl handelt.

```
using System;

public class Primzahl
{
  public static void Main()
  {
    int[] Primzahl = new int[99];
    int i;
    int j;
    bool IstPrimzahl;

    for(i = 0; i < Primzahl.Length; i++)
    {
      Primzahl[i] = i + 2;
    }

    foreach(int Zahl in Primzahl)
    {
      if(Zahl == 2 || Zahl == 3)
      {
        Console.Write("{0} ", Zahl);
        continue;
      }

      IstPrimzahl = true;
      for(j = 2; j <= Math.Sqrt(Zahl); j++)
      {
        if(Zahl % j == 0)
        {
          IstPrimzahl = false;
          break;
        }
      }
      if(IstPrimzahl)
      {
        Console.Write("{0} ", Zahl);
      }
    }
  }
}
```

Code 24: Die foreach-Schleife

Die Syntax der `foreach`-Schleife lautet also

```
foreach(Datentyp Variablenname in Arrayname)
{
  Anweisungsblock;
}
```

Wichtig hierbei ist, dass im Schleifenkopf der Datentyp der Schleifenvariablen angegeben wird, sonst meldet der Compiler einen Fehler.

Sie haben in diesem Beispiel nach `PI` ein weiteres Mitglied der Klasse `Math` kennengelernt: `Sqrt`. Diese Funktion berechnet die Wurzel der als Parameter übergebenen Zahl und gibt sie als Wert des Typs `double` zurück.

10.9 Ausblick

Im nächsten Kapitel werden Sie eine weitere Datenstruktur (struct) und Aufzählungsdatentypen kennen lernen.

11 Strukturen

Im diesem Kapitel werden Sie lernen, was eine Struktur ist, welche Unterschiede zu einem Array bestehen, und was Aufzählungsdatentypen sind.

11.1 Einleitung

Im letzten Kapitel haben Sie Arrays kennengelernt, um große Mengen gleichartiger Daten über eine gemeinsame Variable ansprechen zu können. In diesem Kapitel werden Sie Strukturen kennen lernen, die ebenfalls dazu dienen, Daten zusammenzufassen. Allerdings können Sie hier im Gegensatz zu Arrays auch verschiedene Datentypen mischen.

11.2 Strukturen

Ein einfaches Beispiel, um eine Struktur zu implementieren, ist ein Punkt in einem Koordinatensystem. Er besitzt einen Namen (`string`), eine x- sowie eine y-Koordinate (`double`). In C# werden Strukturen mittels des Schlüsselworts `struct` angelegt, wobei diese Definition außerhalb der `class`-Anweisung erfolgt.

Beachten Sie, dass Sie allen Variablen das Schlüsselwort `public` voranstellen müssen (warum das so ist, werden Sie später noch erfahren).

```
using System;

struct SPunkt
{
  public string Name;
  public double X;
  public double Y;
}

public class Punkt
{
  public static void Main()
  {
    SPunkt EinPunkt;

    Console.Write("Bitte geben Sie den Namen des Punktes
    ein: ");
    EinPunkt.Name = Console.ReadLine();
    Console.Write("Bitte geben Sie die x-Koordinate ein:
    ");
    EinPunkt.X = Double.Parse(Console.ReadLine());
    Console.Write("Bitte geben Sie die y-Koordinate ein:
    ");
    EinPunkt.Y = Double.Parse(Console.ReadLine());

    Console.WriteLine("\nSie haben folgende Informationen
    eingegeben:");
    Console.WriteLine("Name des Punktes: {0}",
    EinPunkt.Name);
    Console.WriteLine("Koordinaten: ({0} | {1})",
    EinPunkt.X, EinPunkt.Y);
  }
}
```

Code 25: Festlegen einer Struktur mittels struct

Dieses Programm legt also zunächst eine Struktur für einen Punkt namens *SPunkt* an. In der Methode `Main` wird dann mittels

```
SPunkt EinPunkt;
```

eine Variable *EinPunkt* vom Typ `SPunkt` angelegt (genauso, wie eine Variable mit einem vorgefertigten Datentyp wie beispielsweise `int` oder `double` angelegt wird). Auf die einzelnen Komponenten von *EinPunkt* können Sie nun zugreifen, indem Sie dem Variablennamen den durch einen Punkt abgetrennten Komponentennamen anschließen.

Die Struktur gibt also sozusagen den Bauplan vor, legt aber selbst noch keine Variablen an. Dies geschieht erst, wenn Sie im Programm Variablen vom Datentyp der Struktur deklarieren. Wenn Sie viele solche Variablen benötigen, können Sie natürlich ein Array von einer Struktur anlegen.

Die Syntax einer Struktur lautet also

```
struct Name
{
  public Datentyp Komponentenname;
}
```

Vielleicht ist Ihnen in dem Programm der Ausdruck

```
Double.Parse(Console.ReadLine())
```

aufgefallen. Er dient dazu, den von `ReadLine` eingelesenen String in einen Double-Wert zu konvertieren. Darauf wird im nächsten Kapitel aber noch genauer eingegangen.

11.3 Arrays von Strukturen

Wenn Sie in einem Programm häufig die selbe Struktur benötigen, können Sie entweder entsprechend viele Variablen deklarieren oder wieder ein Array verwenden.

```
using System;

struct SComputer
{
  public string Name;
  public byte IP_1;
  public byte IP_2;
  public byte IP_3;
  public byte IP_4;
}

public class Computeradressen
{
  public static void Main()
  {
    int i;
    SComputer[] Computer = new SComputer[5];

    for(i = 0; i < 5; i++)
    {
      Console.Write("Bitte geben Sie den Computernamen
       ein: ");
      Computer[i].Name = Console.ReadLine();
      Console.Write("Bitte geben Sie den ersten Teil der
       IP ein: ");
      Computer[i].IP_1 = Byte.Parse(Console.ReadLine());
      Console.Write("Bitte geben Sie den zweiten Teil der
       IP ein: ");
      Computer[i].IP_2 = Byte.Parse(Console.ReadLine());
      Console.Write("Bitte geben Sie den dritten Teil der
       IP ein: ");
      Computer[i].IP_3 = Byte.Parse(Console.ReadLine());
      Console.Write("Bitte geben Sie den vierten Teil der
       IP ein: ");
      Computer[i].IP_4 = Byte.Parse(Console.ReadLine());
    }

    Console.WriteLine();

    for(i = 0; i < 5; i++)
    {
      Console.WriteLine("{0}. Computer: ", i + 1);
      Console.WriteLine("Name: {0}", Computer[i].Name);
      Console.Write("IP: {0}.", Computer[i].IP_1);
      Console.Write("{0}.", Computer[i].IP_2);
      Console.Write("{0}.", Computer[i].IP_3);
      Console.WriteLine("{0}", Computer[i].IP_4);
    }
  }
}
```

Code 26: Arrays von Strukturen

Beachten Sie dabei, dass die einzelnen Werte für die IP-Adresse in Variablen vom Typ `byte` gespeichert werden, also nur Werte zwischen 0 und 255 aufnehmen können. Wird zur Laufzeit des Programms ein Wert außerhalb dieses Bereichs eingegeben (zum Beispiel 300), bricht das Programm mit einer Fehlermeldung ab.

11.4 Verschachtelte Strukturen

Im letzten Beispiel haben Sie eine Struktur erstellt, die den Namen und die IP-Adresse eines Computers speichert. Sie hätten die IP-Adresse aber auch getrennt in einer zweiten Struktur anlegen können und diese dann in der Struktur *SComputer* verwenden. Dieses System nennt man *verschachtelte Strukturen*.

```
struct SIP
{
  public byte IP_1;
  public byte IP_2;
  public byte IP_3;
  public byte IP_4;
}

struct SComputer
{
  public string Name;
  public SIP IP;
}
```

Wenn Sie nun im Programm eine Variable namens *Computer* des Typs `SComputer` anlegen, haben Sie Zugriff auf folgende Komponenten.

```
Computer.Name
Computer.IP.IP_1
Computer.IP.IP_2
Computer.IP.IP_3
Computer.IP.IP_4
```

Wie Sie sehen, können Sie Strukturen also ganz einfach verschachteln, indem Sie sie wiederum durch Punkte trennen. Die einzige Einschränkung, die Sie dabei beachten müssen, ist, dass sich die Strukturen nicht beide gegenseitig enthalten dürfen.

11.5 Der Aufzählungsdatentyp enum

Es gibt in C# eine weitere Möglichkeit, eigene Datentypen zu deklarieren. Mittels des Schlüsselworts `enum` können Sie einen Datentyp deklarieren, dessen Wertebereich nur aus bestimmten, festgelegten Werten besteht. Mit `enum` ist es beispielsweise möglich, einen Datentyp zu erstellen, der Monatsnamen als Werte akzeptiert.

```
enum EMonate {Januar, Februar, ..., Dezember};
```

legt beispielsweise den Datentyp *EMonate* an. Intern wird dabei *Januar* durch eine 0 repräsentiert, Februar durch eine 1, ... Die Programmiersprache C# verwendet hierbei einen sogenannten *nullbasierten Index*, das heißt, dem ersten Wert wird immer die 0 zugewiesen.

In einem Programm können Sie dann eine Variable diesen Typs anlegen und mit den Monatsnamen als Werte hantieren.

```
EMonate Monat;
Monat = Januar;
if(Monat == November)
{
  Console.WriteLine("Bald ist Weihnachten!");
}
```

Sie können den einzelnen Werten aber auch beliebig Zahlen zuweisen, um beispielsweise die Anzahl der Tage je Monat zu speichern.

```
enum EMonate {Januar = 31, Februar = 28, ..., Dezember =
31};
```

Die Syntax des `enum`-Befehls lautet also entweder

```
enum Name {Wert1, Wert2, ...};
```

oder

```
enum Name {Wert1 = Zahl1, Wert2 = Zahl2, ...};
```

Wenn Sie nicht allen Werten explizit Zahlen zuweisen, werden alle nach der letzten Zuweisung folgenden Werte automatisch durchnummeriert. So weist beispielsweise die Zeile

```
enum Winter {Oktober = 10, November, Dezember};
```

dem Wert *Oktober* die Zahl 10 zu. Da für die Werte *November* und *Dezember* keine eigene Zuweisung existiert, werden sie entsprechend nummeriert und intern mit den Zahlen 11 und 12 belegt.

11.6 Ausblick

Im nächsten Kapitel werden Sie sich eingehend mit Strings beschäftigen und ihre Manipulationsmöglichkeiten kennen lernen.

12 Strings

In diesem Kapitel werden Sie sich eingehender mit dem Datentyp `string` und seinen Manipulationsmöglichkeiten beschäftigen. Dieses Kapitel ist ein bisschen anders als die bisherigen Kapitel aufgebaut, denn es enthält sehr viele neue Befehle.

Sie müssen sich nicht alle sofort merken, sondern können immer wieder nachschlagen. Insofern dient dieses Kapitel auch als Referenz.

12.1 Vergleichen von Strings

Wie man zwei Strings mittels der Methode `Compare` miteinander vergleicht, haben Sie bereits im Kapitel über Schleifen kennengelernt. Zur Wiederholung sollten Sie sich folgendes Programmfragment trotzdem noch einmal ansehen.

```
Console.Write("Bitte geben Sie einen Namen ein: ");
string Name1 = Console.ReadLine();
Console.Write("Bitte geben Sie noch einen Namen ein: ");
string Name2 = Console.ReadLine();

int Vergleich = String.Compare(Name1, Name2);
if(Vergleich == 0)
  Console.WriteLine("Die beiden Namen sind gleich!");
else if(Vergleich < 0)
  Console.WriteLine("Der erste Name ist alphabetisch
  kleiner!");
else
  Console.WriteLine("Der erste Name ist alphabetisch
  größer!");
```

So lange Sie nur auf Gleichheit testen wollen, geht es allerdings auch einfacher. Hierzu existiert nämlich die Methode `Equals`, die im Prinzip genau so arbeitet, aber `true` oder `false` zurückgibt, je nachdem, ob die Strings gleich sind oder nicht.

```
if(String.Equals(Name1, Name2))
  Console.WriteLine("Die beiden Namen sind gleich!");
else
  Console.WriteLine("Die beiden Namen sind ungleich!");
```

In dem Kapitel Schleifen haben Sie außerdem gelernt, warum Sie Strings mittels einer Methode vergleichen müssen und das nicht über den Operator `==` funktioniert (weil der Datentyp `string` nämlich ein Referenztyp ist und somit die Variable keinen Wert an sich, sondern nur einen Verweis auf eine Speicherstelle enthält).

12.2 Strings kopieren

Dasselbe Problem tritt auf, wenn Sie Strings kopieren wollen. Wahrscheinlich würden Sie es zunächst folgendermaßen versuchen.

```
string Text1 = "Hallo Welt!";
string Text2 = Text1;
```

Sie würden annehmen, dass Sie nun zwei Variablen *Text1* und *Text2* zur Verfügung haben, auf die Sie unabhängig voneinander zugreifen können. Doch dem ist nicht so, denn wie beim Vergleich haben Sie hier nicht auf den Wert der Variablen, sondern auf die Speicherstelle zugegriffen. Daher enthalten sowohl *Text1* als auch *Text2* den selben Verweis auf die selbe Speicherstelle.

Ändern Sie nun aber den Wert des Textes ab, so ändern sich beide Variablen, da Sie ja beide den selben geänderten String im Speicher referenzieren.

Um Strings wirklich zu kopieren und zwei voneinander unabhängige Variablen zu erhalten, können Sie die Methode `Copy` benutzen.

```
string Text1 = "Hallo Welt!";
string Text2 = String.Copy(Text1);
```

12.3 Substrings

Substrings sind Ausschnitte aus Strings, die aus nur einem, aber auch aus mehreren Zeichen bestehen können. In C# gibt es eine sehr einfache Zugriffsmöglichkeit auf einzelne Zeichen eines Strings. Man kann auf sie nämlich genauso zugreifen wie auf Elemente eines Arrays. Allerdings ist damit nur lesender Zugriff möglich.

```
string Text = "Hallo Welt!";
Console.WriteLine(Text[0]);    // Gibt ein "H" aus
```

Um abzufragen, aus wie vielen Zeichen ein Strings überhaupt besteht, können Sie die Eigenschaft `Length` verwenden. Die Zeilen

```
string Text = "Hallo Welt!";
for(i = 0; i < Text.Length; i++)
{
  Console.Write(Text[i]);
}
```

bewirken also das selbe wie

```
string Text = "Hallo Welt!";
Console.Write(Text);
```

Beachten Sie hierbei bitte, dass `Length` die Anzahl der Zeichen des Strings zurückgibt, der höchste Index zur Adressierung einzelner Zeichen aber (wie bei einem Array) um eins niedriger ist!

Es kommt aber oft auch vor, dass Sie einen Teilstring benötigen, der aus mehr als einem Zeichen besteht. Hierfür existiert die Methode `Substring`. Das folgende Beispiel extrahiert aus dem String *Hallo Welt!* das Wort *Welt*.

```
string Text = "Hallo Welt!";
Console.WriteLine(Text.Substring(6, 4));
```

Der erste Parameter gibt die Stelle an, an der der Substring beginnen soll (beachten Sie, dass das erste Zeichen den Index 0 besitzt!) und der zweite die Länge des Substrings.

Allerdings müssen Sie aufpassen, dass Sie keine Substrings anfordern, die größer als der ursprüngliche String sind oder die über das Ende des ursprünglichen Strings hinausgehen, denn sonst bricht das Programm mit einer Fehlermeldung ab.

Es gibt noch eine weitere Variante der Methode `Substring`, die nur einen Parameter erwartet. Dieser Parameter gibt die Stelle an, an welcher der Substring beginnen soll. Das Ende des Substrings entspricht in diesem Fall dem Ende des zu unterteilenden Strings.

```
string Text = "Hallo Welt!";
Console.WriteLine(Text.Substring(4));
```

Dieses Beispiel liefert den Substring ab dem fünften Zeichen zurück, also den Text *o Welt!*.

12.4 Strings manipulieren

Außer den bisher vorgestellten Befehlen zum Auslesen von Teilstrings gibt es auch Befehle zum schreibenden Zugriff auf Teilstrings. Die erste Variante ist die Methode `Insert`, die einen String an einer bestimmten Position in einen anderen einfügt. Im ersten Parameter wird dabei die Position, im zweiten der einzufügende String übergeben.

```
string Text = "Hallo !";
Text = Text.Insert(6, "Welt");
Console.WriteLine(Text);
```

gibt also *Hallo Welt!* auf dem Bildschirm aus. Ähnlich kann man einen String auch wieder aus einem anderen entfernen, indem man die Methode `Remove` benutzt. Ihr wird als erster Parameter die Startposition und als zweiter die Länge des zu entfernenden Strings übergeben.

```
string Text = "Hallo WeltWelt!";
Text = Text.Remove(6, 4);
```

entfernt demnach das erste Auftauchen des Wortes *Welt* im String *Text*. Außerdem kann man einen String auch komplett in Groß- oder Kleinbuchstaben verwandeln, indem man die Methoden `ToUpper` beziehungsweise `ToLower` verwendet.

```
string Text = "Hallo Welt!";
Console.WriteLine(Text);              // Hallo Welt!
Console.WriteLine(Text.ToUpper());    // HALLO WELT!
Console.WriteLine(Text.ToLower());    // hallo welt!
```

12.5 Durchsuchen von Strings

Schließlich haben Sie in C# auch noch die Möglichkeit, Strings zu durchsuchen. Die einfachste Variante prüft, ob der String mit einem bestimmten Teilstring beginnt, diese Methode heißt `StartsWith` und liefert `true` oder `false` zurück.

```
string Text = "Hallo Welt!";
if(Text.StartsWith("Hallo"))
  Console.WriteLine("Der Text fängt mit \"Hallo\" an!");
else
  Console.WriteLine("Der Text fängt nicht mit \"Hallo\"
  an!");
```

Ganz analog dazu gibt es eine Methode `EndsWith`, die überprüft, ob ein String mit einem bestimmten Teilstring endet. Außerdem gibt es noch die Methode `IndexOf`, um festzustellen, ob und wo ein bestimmter Teilstring in einem String enthalten ist. Als erster Parameter wird dabei der zu suchende Teilstring und als zweiter die Startposition übergeben. Wird der Teilstring gefunden, so gibt die Methode `IndexOf` die Position zurück, andernfalls eine -1.

```
string Text = "Hallo Welt!";
int i = Text.IndexOf("Welt", 0);
if(i != -1)
  Console.WriteLine("Das Wort \"Welt\" ist an Position
  {0} enthalten!", i);
else
  Console.WriteLine("Das Wort \"Welt\" ist nicht in dem
  Text enthalten!");
```

Obwohl C# noch viel mehr Methoden bereithält, um Strings zu manipulieren, kennen Sie jetzt die wichtigsten, die in der Praxis am häufigsten benötigt werden.

12.6 Konvertieren von Strings

Im letzten Kapitel haben Sie einen Ausdruck verwendet, um einen String in einen Wert vom Typ `double` zu konvertieren.

```
Double.Parse(Console.ReadLine())
```

Dieser Ausdruck durchsucht den String, der von `ReadLine` zurückgeliefert wird, nach einer Zahlefolge, die wie ein Wert des Typs `double` aussieht, und - sofern eine solche Zahlenfolge gefunden wird - konvertiert diese in einen Wert des Typs `double` und gibt diesen dann zurück, so dass er beispielsweise in einer Variablen gespeichert werden kann.

Diese explizite Konvertierung ist nicht nur für den Typ double möglich, sondern beispielsweise auch für `int` mittels

```
Int32.Parse(Console.ReadLine())
```

oder für `float` mittels

```
Single.Parse(Console.ReadLine())
```

Hierbei muss man beachten, dass die Klassen für die Datentypen leider nicht immer die erwarteten Namen haben, wie Sie bereits am Datentyp `float` und der dazugehörigen Klasse `Single` sehen können. Die folgende Tabelle listet alle Datentypen mit den zuheörigen Klassen auf.

Datentyp	***Klasse***
bool	Boolean
byte	Byte
char	Char
decimal	Decimal
double	Double
float	Single
int	Int32
long	Int64
short	Int16
uint	UInt32
ushort	UInt16
ulong	UInt64

Tabelle 9: Datentypen und ihre Klassen

Beachten Sie bitte außerdem, dass Sie statt `Console.ReadLine` natürlich jeden beliebigen Stringausdruck verwenden können.

12.7 Ausblick

Im nächsten Kapitel werden Sie lernen, was Funktionen beziehungsweise Methoden sind.

13 Methoden

In diesem Kapitel werden Sie lernen, was Funktionen beziehungsweise Methoden sind.

13.1 Einleitung

In den vergangenen Kapiteln haben Sie die grundlegenden Sprachelemente von C# kennen gelernt. Doch bis jetzt haben Sie in jedem Programm, das Sie entwickelt haben, mindestens zwei Zeilen geschrieben, deren Bedeutung Sie noch nicht kennen:

```
public class ProgrammName
```

und

```
public static void Main()
```

Mit diesen beiden Zeilen werden Sie sich in diesem und dem nächsten Kapitel beschäftigen.

13.2 Funktionen

Oft kommt es in Programmen vor, dass Sie eine bestimmte Funktionalität öfter benötigen. Sie können bisher nichts anderes machen, als den entsprechenden Codeblock noch einmal zu schreiben oder zu kopieren. Dadurch werden große Programme sehr schnell unübersichtlich und fehleranfällig, da Sie unter Umständen für eine Änderung mehrere Programmteile ändern müssten.

Dieses Problem lässt sich durch Funktionen (oder Methoden) lösen. Funktionen sind hierbei der allgemeinere Begriff, Methoden sind eine besondere Art von Funktionen. Im nächsten Kapitel werden Sie den Unterschied kennen lernen. Manchmal wird statt Funktion oder Methode auch der etwas altmodische Begriff Prozedur verwendet.

Stellen Sie sich Funktionen einfach als eigenständige Programmabschnitte vor, die Sie aus Ihrem Programm heraus aufrufen können - und zwar beliebig oft. Wenn Sie an einer Funktion etwas ändern müssen, müssen Sie dies nur einmal tun, da alle anderen Programmteile ja immer nur auf diese eine Funktion zurückgreifen. Dies erleichtert das Testen und Fehler auffinden erheblich. Außerdem können Sie Funktionen in mehreren Programmen einsetzen und können so auf bereits geschriebenen (und getesteten) Code zurückgreifen.

Wie man in C# eine Funktion erstellt, soll das folgende Beispiel verdeutlichen. Das Programm enthält eine Funktion namens Berechnen, welche die Wurzel der Zahl 2 berechnet und das Ergebnis anschließend auf den Bildschirm ausgibt.

```
using System;

public class Wurzel
{
  public static void Main()
  {
    Console.WriteLine("Dieses Programm berechnet die
    Wurzel von 2.");
    Berechnen();
    Console.WriteLine("Dazu benutzt es die Funktion
    Berechnen.");
  }

  public static void Berechnen()
  {
    double Wurzel;
    Wurzel = Math.Sqrt(2);
    Console.WriteLine("Die Wurzel von 2 ist " + Wurzel +
    ".");
    return;
  }
}
```

Code 27: Wurzelberechnung durch Aufruf einer Funktion

Dieses Programm gibt zunächst den Text *Dieses Programm berechnet die Wurzel von 2.* auf dem Bildschirm aus und ruft dann die Funktion `Berechnen` auf. Die eigentliche Funktion Berechnen wird weiter unten im Programm definiert, hinter der schließenden geschweiften Klammer des Bereichs, die bisher das Ende eines Programms markiert hat.

Die Zeile

```
public static void Berechnen()
```

gibt den Namen der Funktion an, nämlich Berechnen, und der Funktionscode wird in den darauf folgenden Klammern eingeschlossen.

Aufgerufen wird eine selbst entwickelte Funktion mittels ihres Namens, gefolgt von einem Paar runder Klammern. Diese Syntax entspricht genau der Syntax, die Sie schon von einigen C#-Befehlen her kennen, beispielsweise

```
Console.WriteLine();
```

Prinzipiell kann ein Programm beliebig viele Funktionen enthalten, sie müssen einfach nur untereinander definiert werden. Wichtig dabei ist, dass eine Funktion nicht innerhalb einer anderen definiert werden kann, sondern immer nur nach dem Ende der vorhergehenden Funktion. Die Reihenfolge der Funktionen in dem Programm spielt allerdings keine Rolle.

Auch wenn Funktionen nicht verschachtelt definiert werden dürfen, so können sie durchaus verschachtelt aufgerufen werden, das heißt, Sie rufen eine Funktion A auf, die ihrerseits eine Funktion B aufruft, die vielleicht wiederum auf Funktion C zurückgreift. Hierbei sind Ihnen keine Grenzen gesetzt. Es gibt sogar Funktionen, die sich selbst aufrufen, sogenannte *rekursive Funktionen*, doch dazu später mehr.

Beachtenswert ist noch die letzte Zeile in der Funktion `Berechnen`:

```
return;
```

Mit dieser Anweisung wird der Computer angewiesen, die Funktion zu verlassen und an die Stelle im Programm zurückzuspringen, von der aus die Funktion aufgerufen wurde.

Die `return`-Anweisung ist allerdings optional, wenn sie die letzte Anweisung innerhalb der Funktion ist. Sie könnte in diesem Beispiel also auch weggelassen werden.

13.3 Lokale Variablen

Eine Besonderheit von C# ist in dem letzten Beispiel noch gar nicht aufgefallen, jede Funktion erhält nämlich einen komplett eigenen Satz von Variablen. So können Sie beispielsweise in jeder Funktion eine eigene (!) Variable namens *i* definieren, und unabhängig voneinander darauf zugreifen. Wenn Sie den Wert von *i* in der ersten Funktion verändern, ändert dies nicht den Wert von *i* in der zweiten Funktion, da es sich um vollständig unterschiedliche Variablen handelt, die nur den Namen gemeinsam haben.

```
public void FunktionA
```

Auf der anderen Seite bedeutet das, dass Sie nicht ohne weiteres auf eine Variable in einer anderen Funktion zugreifen können. Man spricht in diesem Zusammenhang von der *Sichtbarkeit* von Variablen, da sie im Prinzip immer nur in der Funktion verwendbar (oder eben sichtbar) sind, in der sie definiert wurden. Daher bezeichnet man solche Variablen auch als *lokale Variablen*.

Das Gegenteil von lokalen sind *globale Variablen*, die es in C# aber nicht gibt. Lokale Variablen haben vor allem den Vorteil, dass man nicht aus Versehen eine Variable in einem anderen Programmteil verändern kann, indem man eine gleich benannte Variable verwendet. Dies vermindert die Anzahl der Fehlerquellen und vereinfacht die Fehlersuche erheblich.

13.4 Parameterübergabe

Trotzdem kann es natürlich vorkommen, dass man zwischen zwei Funktionen Daten austauschen muss. Da dies wegen der lokalen Sichtbarkeit von Variablen nicht so ohne weiteres möglich ist, kann man einer Funktion einen oder mehrere Parameter oder Argumente übergeben.

Diese werden in den runden Klammern hinter dem Funktionsnamen übergeben.

```
using System;

public class Primzahl
{
  public static void Main()
  {
    int Zahl;

    Console.Write("Bitte geben Sie eine ganze Zahl größer
    als 1 ein: ");
    Zahl = Int32.Parse(Console.ReadLine());
    PrimzahlTest(Zahl);
  }

  public static void PrimzahlTest(int Nummer)
  {
    bool IstPrimzahl;

    if(Nummer < 2)
      IstPrimzahl = false;
    else if(Nummer == 2 || Nummer == 3)
      IstPrimzahl = true;
    else if(Nummer % 2 == 0)
      IstPrimzahl = false;
    else
    {
      IstPrimzahl = true;
      for(int i = 2; i <= Math.Sqrt(Nummer); i++)
      {
        if(Nummer % i == 0)
        {
          IstPrimzahl = false;
          break;
        }
      }
    }

    if(IstPrimzahl)
      Console.WriteLine("{0} ist eine Primzahl!", Nummer);
    else
      Console.WriteLine("{0} ist keine Primzahl!", Nummer);

    return;
  }
}
```

Code 28: Funktionen und Parameterübergabe

In diesem Beispiel wird der Benutzer aufgefordert, eine Zahl einzugeben, die dann in einen `int`-Wert umgewandelt wird und in der Variablen *Zahl* gespeichert wird. Diese Variable wird dann beim Aufruf der Funktion `PrimzahlTest` übergeben, indem der Variablenname in die runden Klammern geschrieben wird. Bei der Definition der Funktion `PrimzahlTest` wiederum wurde angegeben, dass sie ein Argument vom Typ `int` erwartet, und dass dieses Argument innerhalb der Funktion `PrimzahlTest` über die Variable *Nummer* angesprochen werden soll.

Beim Aufruf der Funktion `PrimzahlTest` passiert also folgendes: Der Compiler stellt fest, dass die Variable *Zahl* übergeben werden soll und innerhalb der aufgerufenen Funktion *Nummer* heißen soll. Daher kopiert er den Wert der Variablen *Zahl* in die Variable *Nummer* und startet die Funktion `PrimzahlTest`. Diese kann nun auf den Wert der übergebenen Variablen zugreifen.

Allerdings findet dieser Zugriff nur auf einer Kopie der ursprünglichen Variable statt. Verändert die aufgerufene Funktion also den Wert der Variablen, so hat dies keinerlei Auswirkungen auf die Variable im aufrufenden Programm.

Außerdem sollte man noch beachten, dass bei der Definition der Funktion der Typ der erwarteten Variable angegeben wird (in diesem Fall `int`), damit überprüft werden kann, ob ein übergebenes Argument zulässig ist. Der Typ wird beim Aufruf der Funktion aber nicht mehr mit angegeben.

Selbstverständlich kann einer Funktion auch mehr als ein Argument übergeben werden. Die verschiedenen Argumente werden dabei sowohl in der Definition als auch beim Aufruf durch Kommata getrennt. Ein Beispiel hierfür liefert die folgende Funktion, der zwei Seiten eines rechtwinkligen Dreiecks übergeben werden (um genau zu sein, die beiden Seiten, die den 90°-Winkel einschließen), und die dann die Länge der dritten Seite mittels des Satzes von Pythagoras ($a^2 + b^2 = c^2$) berechnet.

```
using System;

public class Dreieck
{
  public static void Main()
  {
    double a, b;

    Console.Write("Bitte geben Sie die Länge der Seite a ein:
");
    a = Double.Parse(Console.ReadLine());
    Console.Write("Bitte geben Sie die Länge der Seite b ein:
");
    b = Double.Parse(Console.ReadLine());

    Pythagoras(a, b);
  }

  public void Pythagoras(double a, double b)
  {
    double c = Math.Sqrt(a * a + b * b);
    Console.WriteLine("Die dritte Seite ist {0} lang!", c);
    return;
  }
}
```

Code 29: Übergabe mehrerer Argumente an die Funktion Pythagoras

Allerdings müssen die Typen nicht alle identisch sein, wenn Sie einer Funktion mehrere Parameter übergeben. Sie können also auch ohne weiteres eine Funktion entwickeln, die einen `int` und einen `string` erwartet. Sie müssen nur darauf achten, dass beim Aufruf Variablen des richtigen Typs übergeben werden, und zwar in genau der richtigen Reihenfolge, die bei der Definition der Funktion angegeben wurde.

13.5 Rückgabewert

Manchmal soll eine Funktion aber nicht nur Argumente übergeben bekommen, sondern unter Umständen auch einen Wert zurückgeben. Ein Beispiel für eine solche Funktion, die in C# bereits enthalten ist, ist die Wurzelfunktion `Math.Sqrt`, die in dem Beispiel am Anfang dieses Kapitels eingesetzt wurde.

Math.Sqrt erwartet ein Argument vom Typ double, und gibt ebenfalls einen Wert vom Typ double zurück, nämlich die berechnete Wurzel der übergebenen Zahl. Diesen Wert bezeichnet man in C# als Rückgabewert. Eine Funktion kann jeden beliebigen Typ als Rückgabewert verwenden. Sie können also nicht nur beispielsweise int, double oder string zurückgeben, sondern sogar selbstdefinierte Typen.

Der Typ des Rückgabewerts wird bei der Definition direkt vor dem Funktionsnamen angegeben, beispielsweise also:

```
using System;

public class Dreieck
{
  public static void Main()
  {
    double a, b, c;

    Console.Write("Bitte geben Sie die Länge der Seite a ein:
");
    a = Double.Parse(Console.ReadLine());
    Console.Write("Bitte geben Sie die Länge der Seite b ein:
");
    b = Double.Parse(Console.ReadLine());

    c = Pythagoras(a, b);
    Console.WriteLine("Die dritte Seite ist {0} lang!", c);
  }

  public double Pythagoras(double a, double b)
  {
    double c = Math.Sqrt(a * a + b * b);
    return c;
  }
}
```

Code 30: Modifizierte Funktion Pythagoras

Dieses leicht abgewandelte Beispiel gibt einen Wert vom Typ `double` zurück. Damit dies tatsächlich funktioniert, sind zwei kleine Änderungen am Programm nötig: Zum einen muss das aufrufende Programm den Rückgabewert irgendwie entgegennehmen, indem es ihn beispielsweise in einer Variablen speichert.

```
c = Pythagoras(a, b);
```

Zum anderen muss die Funktion `Pythagoras` einen Wert an den Aufrufer zurückgeben:

```
return c;
```

Mittels der `return`-Anweisung wird also nicht nur die Funktion beendet, sondern auch der Rückgabewert festgelegt. Beachten Sie, dass man die ganze Funktion `Pythagoras` in diesem Beispiel auf eine Zeile kürzen könnte, da die Zwischenspeicherung in der Variablen *c* eigentlich überflüssig ist:

```
return Math.Sqrt(a * a + b * b);
```

Allerdings gibt es Funktionen, die keinen Wert zurückgeben, wie beispielsweise alle Funktionen, die Sie bisher in diesem Kapitel entwickelt haben (oder wie beispielsweise die Funktion `Console.WriteLine`, die ebenfalls nur etwas ausführt, aber nichts zurückgibt). Bei diesen Funktionen muss man als Typ des Rückgabewerts `void` angeben.

13.6 Call by value / call by reference

Vorhin wurde erwähnt, dass - sobald eine Funktion aufgerufen wird - die Werte der zu übergebenden Variablen in die lokalen Variablen der Funktion kopiert werden. Dies stimmt jedoch nur bedingt, nämlich bei den sogenannten einfachen Datentypen wie `int`, `double` oder `char`.

Diese Übergabe von Argumenten wird *call by value* genannt, da der Wert einer Variablen übergeben wird.

Bei Referenzdatentypen - wie beispielsweise Strings oder Arrays - hätte call by value aber zwei eklatante Nachteile, denn erstens würde durch das Kopieren unter Umständen sehr viel Speicher belegt und zweitens könnte das Kopieren bei großen Datenmengen viel Zeit in Anspruch nehmen. Deshalb wird bei solchen Datentypen nur eine Referenz auf eine Speicheradresse übergeben. Beide Funktionen greifen dann auf die selben Daten zu, auch wenn die Variablen, mittels derer zugegriffen wird, eventuell anders heißen.

Ändert also eine Funktion den Wert einer Variablen, die als Referenz übergeben wurde, so ändert sich auch der Wert der Variablen im aufrufenden Programm. Diese Art der Übergabe von Argumenten wird *call by reference* genannt. Die folgende Tabelle führt auf, welcher Datentyp standardmäßig wie übergeben wird.

call by value	***call by reference***
bool byte char decimal double enum float int long sbyte short struct uint ulong ushort	array class object string

Tabelle 10: call by value / call by reference

Manchmal kann es vorteilhaft sein, auch einen einfachen Datentyp als Referenz zu übergeben, beispielsweise, wenn die Funktion explizit dessen Wert ändern soll. Dies erreicht man in C# dadurch, dass bei der Funktionsdefinition vor dem Datentyp das Schlüsselwort `ref` angegeben wird.

Die folgende Funktion erhält beispielsweise zwei Zahlen als Argumente, quadriert diese und gibt das Ergebnis nicht mittels `return` zurück (denn mittels `return` kann ja nur ein Ergebnis zurückgegeben werden), sondern, indem es die Ergebnisse in den ursprünglichen Variablen speichert.

```
public void Quadrieren(ref int a, ref int b)
{
  a *= a;
  b *= b;
}
```

13.7 Die Funktion Main

Wie Sie inzwischen vielleicht schon selbst ahnen, handelt es sich auch bei dem Programmabschnitt, der durch

```
public static void Main()
```

eingeleitet wird, um eine Funktion. Allerdings ist die Main-Funktion eine ganz besondere Funktion, denn nach ihr sucht das Betriebssystem, sobald das Programm gestartet wird. Deshalb muss sie auch in jedem Programm enthalten sein. Allerdings kann man auch ihr Argumente übergeben und einen Rückgabewert definieren.

Bisher haben Sie als Rückgabewert immer `void` angegeben, alternativ können Sie in C# auch `int` verwenden. Sie können dann die `return`-Anweisung nutzen, um einen Rückgabewert an den Aufrufer (das Betriebssystem) zurückzugeben, der beispielsweise unter der DOS-Eingabeaufforderung mittels der Variablen errorlevel ausgewertet werden kann. Diese Technik wird oft dazu verwendet, zu signalisieren, ob das Programm ordnungsgemäß abgearbeitet wurde. Meldet ein Programm 0 zurück, ist alles in Ordnung, jede andere Zahl beschreibt einen Fehler.

Das folgende Programmbeispiel fordert den Benutzer auf, zwei Zahlen einzugeben, addiert diese, und gibt das Ergebnis mittels return an das Betriebssystem zurück.

```
using Systen;

public class Addieren
{
  public static int Main()
  {
    int a, b;

    Console.Write("Bitte geben Sie eine Zahl ein: ");
    a = Int32.Parse(Console.ReadLine());
    Console.Write("Bitte geben Sie noch eine Zahl ein: ");
    b = Int32.Parse(Console.ReadLine());

    return a + b;
  }
}
```

Code 31: Die Funktion Main

Außerdem können der Main-Funktion auch Argumente übergeben werden. Diese werden unter C# als Array von Strings behandelt und müssen gegebenenfalls in einen passenden Datentyp konvertiert werden. Beispielsweise kann man das vorangegangene Programm so abändern, dass es die beiden Zahlen als Argumente erwartet.

```
using Systen;

public class Addieren
{
  public static int Main(string[] Argumente)
  {
    int a, b;

    a = Int32.Parse(Argumente[0]);
    b = Int32.Parse(Argumente[1]);

    return a + b;
  }
}
```

Code 32: Die Funktion Main und erwartete Argumente

Ruft man dieses Programm von der Kommandozeile nun mittels der Zeile

```
Addieren 17 23
```

auf, enthält das Stringarray *Argumente* zwei Einträge, nämlich die Strings (!) 17 und 23. Diese lassen sich aber problemlos in den Datentyp int konvertieren und dann weiterverarbeiten. Der Name der Variablen *Argumente* ist übrigens beliebig.

Wie Ihnen vielleicht schon aufgefallen ist, haben Sie in diesem Kapitel noch nichts über die beiden Schlüsselwörter `public` und `static` erfahren, die in den Funktionsdefinitionen aufgetaucht sind. Mehr über diese beiden Schlüsselwörter werden Sie erfahren, sobald Sie Klassen kennen gelernt haben.

13.8 Ausblick

Im nächsten Kapitel werden Sie lernen, was Klassen sind, und was man unter objektorientierter Programmierung versteht.

14 Klassen

In diesem Kapitel werden Sie lernen, was Klassen sind, und was man unter objektorientierter Programmierung versteht.

Da Klassen ein äußerst umfangreiches Thema darstellen, aber grundlegend für das weitere Verständnis sind, ist dieses Kapitel speziell im Hinblick auf Einsteiger etwas umfangreicher und ausführlicher ausgefallen. Falls Sie nicht alles problemlos auf Anhieb verstehen, so macht das nichts, dies ist am Anfang bei der objektorientierten Programmierung völlig normal.

Lassen Sie sich daher für dieses Kapitel ruhig etwas mehr Zeit und arbeiten Sie die einzelnen Abschnitte mehrmals durch, bis Sie glauben, sie wirklich verstanden zu haben.

14.1 Einleitung

Alle Programme, die Sie bisher geschrieben haben, waren prozedural aufgebaut, das heißt, sie bestanden aus einer oder mehreren Funktionen, die sich gegenseitig aufgerufen haben.

Allerdings hatten alle diese Programme eins gemeinsam: Der Hauptaspekt bei der Programmierung lag auf den Anweisungen. Dieses Programmierparadigma hat aber einen entscheidenden Nachteil, denn es geht am eigentlichen Grund, warum Programme überhaupt geschrieben werden, vorbei.

Programme werden nämlich geschrieben, um Daten zu verarbeiten. Die Anweisungen, mit denen dies geschieht, sollten eigentlich nur sekundär sein. Aber als Programmierer beschäftigt man sich hauptsächlich mit den Anweisungen und nicht mit den Daten. Ein prozedurales Programmierparadigma ist deshalb nicht sehr natürlich.

Um dieses Problem zu lösen, entstand in den neunziger Jahren die objektorientierte Programmierung (OOP). Bei der OOP werden die Daten mit den speziell zu ihnen gehörenden Funktionen in einen gemeinsamen Behälter, einem sogenannten Objekt, gespeichert.

Ein Objekt enthält also Variablen, um die Daten aufzunehmen, und Funktionen, um diese Daten zu bearbeiten. Die Daten in einem Objekt können dann nur noch von den Funktionen verarbeitet werden, die zum Objekt gehören. Außerdem können die Funktionen auch nur noch mit genau den Daten, für die entworfen wurden, verwendet werden. Dieses Verfahren der Integration nennt man in der OOP "kapseln".

Die Kapselung ist eins der tragenden Konzepte der objektorientierten Programmierung und trägt vor allem zur Fehlerverminderung bei, da auf Daten nicht mehr beligbig, sondern nur noch mit ausgewählten und (hoffentlich) fehlerfreien Funktionen zugegriffen werden kann.

Im letzten Kapitel haben Sie bereits gelernt, was Funktionen eigentlich sind, allerdings wurde immer wieder darauf hingewiesen, dass es auch sogenannte Methoden gibt. Methoden sind Funktionen, die zu einem konkreten Objekt gehören und für sich alleine keinen Existenzgrund besitzen. Da in C# alle Funktionen zu einem Objekt gehören (sogar die Main-Funktion in gewissem Sinne, wie Sie noch sehen werden), spricht man in C# im Allgemeinen fast nur von Methoden.

Um diese abstrakte Erklärung des Begrifss Objekt ein bisschen zu veranschaulichen, können Sie sich beispielsweise einen Roboter vorstellen, der durch ein Labyrinth fährt und versucht, den Ausgang zu finden. Dieser Roboter besitzt bestimmte Eigenschaften, wie beispielsweise seine aktuelle Position. Diese Information stellt die Daten des Roboters dar.

Außerdem besitzt er noch bestimmte Fähigkeiten, er kann beispielsweise fahren oder sich drehen. Sobald sich der Roboter bewegt, verändert er natürlich seine Position. Da nur er weiß, wie er sich bewegt, sollte auch nur er seine gespeicherte Position verändern können. Dieser Roboter stellt anschaulich gesprochen ein Objekt dar.

14.2 Einführung in Klassen

Bevor man mit Objekten arbeiten kann, muss man aber festlegen, welche Variablen (in der OOP sagt man *Datenelemente*) und Methoden das Objekt beinhalten soll. Man benötigt sozusagen einen Datentyp für das Objekt. Dieser Datentyp heißt *Klasse*. In der Klassendefinition wird also zunächst genau festgelegt, welche Variablen und Methoden ein Objekt dieser Klasse enthält, aber allein durch die Definition einer Klasse werden noch keine Variablen im Speicher angelegt. Die Klasse ist nämlich nur ein Konstruktionsplan für Objekte.

Um im vorigen Beispiel zu bleiben: Der Roboter existiert nicht einfach so, sondern er wurde nach einem bestimmten Bauplan (eben der *Klasse*) entworfen. Der Bauplan bestimmt, welche Aktionen der Roboter ausführen kann und welche Daten er speichern kann, aber bloß durch die Tatsache, dass ein Bauplan existiert, gibt es noch keinen entsprechenden Roboter.

Objekte sind also konkrete Instanzen einer Klasse. Diesen Unterschied zwischen Klassen und Objekten zu verstehen, ist sehr wichtig, da auf diesen beiden Begriffen die ganze OOP aufbaut.

Das objektorientierte Paradigma ist dabei wesentlich natürlicher als das prozedurale, denn mit Hilfe von Objekten lassen sich reale Dinge sehr gut beschreiben. Denn wie Objekte haben auch reale Dinge bestimmte Eigenschaften (die in den Datenelementen eines Objekts gespeichert werden) und es gibt Aktionen, was man mit ihnen machen kann (dies entspricht den Methoden eines Objekts). Objekte eignen sich daher besser als alle anderen Strukturen einer Programmiersprache, um reale Dinge zu beschreiben, durch sie entsteht ein natürlicher Zugang.

Im ersten größeren Beispiel soll eine Firma objektorientiert beschrieben werden. Dieses Beispiel ist nochmals sehr anschaulich und einfach. Als schließlich drittes und abstrakteres Beispiel werden Sie eine Klasse entwickeln, mit der Sie komplexe Zahlen darstellen und mit ihnen rechnen können.

Stellen Sie sich vor, dass Sie eine Firma gründen. Diese Firma, die außer Ihnen vielleicht nur aus einigen wenigen Mitarbeitern besteht, lässt sich gut mit einem Programm vergleichen, das nur aus einer einzigen Funktion besteht. Alles ist noch sehr übersichtlich, jeder ist für alles zuständig und es gibt keine Struktur.

Doch sobald Ihre Firma größer wird und aus mehr Mitarbeitern besteht, werden Sie sie in Funktionsbereiche gliedern. Die einzelnen Mitarbeiter erhalten eigene Bereiche, für die sie zuständig sind. Trotzdem kann jeder noch auf die Daten und Informationen eines anderen ungehindert zugreifen. Dieses Modell lässt sich mit dem prozeduralen Paradigma vergleichen, in dem Funktionen ein Programm gliedern. Trotzdem darf jede Funktion alles und es gibt keine Zugriffssicherheit.

Wenn Ihre Firma aber irgendwann noch größer ist, werden Sie die vorherigen Funktionsbereiche sicherlich in Abteilungen umwandeln, die zwar alle noch miteinander kommunizieren, aber eine Abteilung darf nicht mehr ungehindert auf die Daten einer anderen zugreifen. Dieses Modell entspricht der OOP.

Die Klasse ist in diesem Fall die Abteilung, sie enthält Datenelemente wie den Namen der Abteilung, die Anzahl der Mitarbeiter, den Vorgesetzten und natürlich die Daten der Abteilung. Außerdem enthält sie Methoden, um beispielsweise Daten anzufordern oder eine Nachricht an die Abteilung zu senden. Jede Abteilung (Vertrieb, Management, ...) ist dann ein Objekt der Klasse *Abteilung*, da es sich um eine konkrete Instanz mit Werten in den Datenelementen handelt.

Zwar hat jede Abteilung einen anderen Namen und eventuell auch unterschiedlich viele Mitarbeiter, aber trotzdem haben sie alle die gleiche, gemeinsame Struktur, die durch die Klasse festgelegt wurde. Die Klasse *Abteilung* ist also wieder der Bauplan, nach dem konkrete Modelle (in diesem Fall die konkreten Abteilungen) erstellt werden.

Der Zugriff auf die Daten findet nur gesichert statt, so kann das Management beispielsweise nicht direkt auf die Informationen des Vertriebs zugreifen und versehentlich etwas verändern, aber dafür gibt es Methoden, die vom Management benutzt (aufgerufen) werden können, um die gewünschten Daten des Vertriebs auf kontrollierte Art und Weise anzufordern und zu erhalten.

Wie bereits gesagt, der bloße Bauplan einer Abteilung garantiert noch nicht, dass auch eine reale Abteilung besteht. Er gibt vielmehr nur an, wie eine solche auszusehen hätte. Deswegen existieren - so lange nur eine Klasse definiert ist - auch noch keine Daten im Speicher. Erst, wenn ein Objekt der Klasse angelegt wird (also ein Modell nach dem Bauplan erstellt wird), existiert eine reale Instanz, die dann natürlich ihre Daten speichern muss und daher auch Speicherplatz belegt.

Ein Vorteil der OOP ist unter anderem, dass keine Abteilung die interne Struktur einer anderen zu kennen braucht, da sie nur über die bereitgestellten Methoden auf die andere zugreift. Wenn sich an der internen Struktur einer Abteilung einmal etwas ändern sollte, müssen nur die Zugriffsfunktionen angepasst werden, die eigentliche Veränderung bleibt für Außenstehende verborgen.

Sie sollten nun ein grundlegendes Verständnis der Begriffe "Klasse" und "Objekt" haben, da die beiden Begriffe im folgenden Beispiel nicht mehr näher erläutert und als bekannt vorausgesetzt werden. Falls Ihnen jetzt noch nichts alles ganz klar ist, sollten Sie den bisherigen Teil noch einmal durcharbeiten, bevor Sie weitermachen.

14.3 Komplexe Zahlen

Wie bereits gesagt geht es im zweiten Beispiel darum, eine Klasse für komplexe Zahlen (den Zahlenraum ℂ) zu entwickeln. Falls Sie wissen, was komplexe Zahlen sind und wie man mit ihnen rechnet, können Sie die nächsten drei Absätze getrost überspringen. Falls Sie aber noch nicht (oder nicht mehr) wissen, was komplexe Zahlen sind, brauchen Sie sich keine Sorgen zu machen, denn ein tiefergehendes Verständnis von komplexen Zahlen ist für das folgende Beispiel nicht notwendig, die kurze Einführung sollten Sie aber dennoch lesen.

Unter den reellen Zahlen (im Zahlenraum ℝ) gibt es bekanntermaßen keine Lösung der Gleichung $x^2 = -1$, denn egal welche Zahl man quadriert, das Ergebnis ist immer größer oder gleich Null und kann somit nicht -1 sein. Um diese Gleichung doch lösen zu können, haben Mathematiker (vereinfacht gesagt) eine Zahl erfunden, die diese Gleichung löst - nämlich die imaginäre Zahl *i*, die als Wurzel von -1 definiert ist.

Komplexe Zahlen sind nun aus zwei reellen Zahlen zusammengesetzt, sie bestehen nämlich aus einem sogenannten Real- und einem Imaginärteil (die im Folgenden mit a und b bezeichnet werden). Der Imaginärteil ist dabei ein Koeffizient (also ein Faktor, mit dem multipliziert wird) der imaginären Zahl i. Eine komplexe Zahl hat demnach die Form $a + b * i$. Auch wenn komplexe Zahl schwer vorstellbar sind, kann man sie beispielsweise sehr gut in einem Koordinatensystem darstellen, indem man den Realteil auf der x-Achse und den Imaginärteil auf der y-Achse einträgt. Der Punkt, den man dabei erhält, stellt die komplexe Zahl grafisch dar.

Angewendet werden komplexe Zahlen beispielsweise in der Computergrafik, um die Koordinaten einzelner Pixel zu speichern.

Da eine einzelne komplexe Zahl ein Objekt darstellt, benötigt man zunächst eine Klasse, die die Eigenschaften und Methoden von komplexen Zahlen festlegt. Um ünerhaupt sinnvoll nutzbar zu sein, muss die Klasse auf jeden Fall zwei Datenelemente enthalten, um den Real- und den Imaginärteil abzuspeichern.

```
public class CKomplexeZahl
{
  private double Real;
  private double Imaginaer;
}
```

Wie Sie sehen, wird eine Klasse mittels des Schlüsselwortes class definiert, gefolgt von dem Klassennamen, in diesem Fall also *CKomplexeZahl*. In ihrem Körper werden zwei Datenelemente vom Typ `double` definiert, *Real* und *Imaginaer*, um die reale und die imaginäre Komponente einer komplexen Zahl zu speichern.

Wie vorhin bereits angedeutet, kann auf die Daten eines Objekts nur mit den dazugehörenden Methoden zugegriffen werden. Dieser Zugriffsschutz muss allerdings für jedes Datenelement einzeln gesetzt werden. Damit ein Datenelement also nicht mehr von außerhalb der Klasse zugänglich ist, muss es mit dem Schlüsselwort `private` gekennzeichnet werden.

Das Gegenteil von `private` stellt das Schlüsselwort `public` dar. Datenelemente, die mit `public` gekennzeichnet sind, sind von außen zugänglich und können von überall verändert werden, allerdings ist dies nur in seltenen Ausnahmefällen sinnvoll.

Die Klasse soll aber insgesamt von außerhalb der Klasse zugänglich sein, daher muss sie als `public` gekennzeichnet werden (sonst könnte man nicht einmal Objekte von ihr instanziieren).

Um es noch einmal zu wiederholen: Diese Klasse an sich legt noch keinen Speicherplatz für Datenelemente an! Sie dient lediglich als Blaupause für Objekte vom Typ `CKomplexeZahl`, die dann jeweils eine konkrete, komplexe Zahl repräsentieren und Speicherplatz für jeweils genau eine komplexe Zahl belegen.

Da die Datenelemente nun aber mittels `private` vor Zugriff von außen geschützt sind, benötigt man Methoden, um überhaupt auf sie zugreifen zu können. Da auf diese Methoden natürlich von außen zugegriffen werden muss, werden diese wiederum als `public` definiert. Außerdem wird noch eine weitere Methode definiert, um eine komplexe Zahl formatiert auszugeben.

```
using System;

public class CKomplexeZahl
{
  private double Real;
  private double Imaginaer;

  public void SetReal(double a)
  {
    Real = a;
  }

  public void SetImaginaer(double b)
  {
    Imaginaer = b;
  }

  public double GetReal()
  {
    return Real;
  }

  public double GetImaginaer()
  {
    return Imaginaer;
  }

  public void WriteLine()
  {
    Console.WriteLine("{0} + {1} * i", Real, Imaginaer);
  }
}
```

Code 33: Die Klasse CKomplexeZahl

Nun kann die Klasse *CKomplexeZahl* bereits die Werte komplexer Zahlen speichern, verändern und die Zahl auf den Bildschirm ausgeben. Zu einem lauffähigen Programm fehlen aber noch die Möglichkeit, Objekte dieser Klasse überhaupt anzulegen, zum anderen fehlt noch eine Main-Methode.

Wie man Objekte anlegt, wissen Sie prinzipiell schon, denn Sie haben schon mit Objekten in C# gearbeitet, allerdings ohne es zu wissen. Die Referenzdatentypen String und Array sind intern nämlich nichts anderes als Klassen (`System.String` beziehungsweise `System.Array`). Um also eine komplexe Zahl anzulegen, gehen Sie genauso vor, wie wenn Sie ein Array anlegen wollten.

```
CKomplexeZahl EineZahl = new CKomplexeZahl();
```

statt

```
int[] EinArray = new int[30];
```

Mit der ersten Zeile wird ein Objekt vom Typ `CKomplexeZahl` angelegt und in der Variablen *EineZahl* gespeichert. Nach dem folgenden Beispielprogramm wird noch einmal genauer auf die Syntax eingegangen. Die noch fehlende Main-Methode wird im Programm in einer weiteren Klasse definiert. Dort werden dann zwei komplexe Zahlen angelegt, ihre Werte gesetzt und schließlich ausgegeben.

```
using System;

public class KomplexeZahlenTest
{
  public static void Main()
  {
    CKomplexeZahl ErsteZahl = new CKomplexeZahl();
    CKomplexeZahl ZweiteZahl = new CKomplexeZahl();

    ErsteZahl.SetReal(5);
    ErsteZahl.SetImaginaer(3.7);
    ZweiteZahl.SetReal(-10.3);
    ZweiteZahl.SetImaginaer(3.2);

    ErsteZahl.WriteLine();
    ZweiteZahl.WriteLine();
  }
}

public class CKomplexeZahl
{
  private double Real;
  private double Imaginaer;

  public void SetReal(double a)
  {
    Real = a;
  }

  public void SetImaginaer(double b)
  {
    Imaginaer = b;
  }

  public double GetReal()
  {
    return Real;
  }

  public double GetImaginaer()
  {
    return Imaginaer;
  }

  public void WriteLine()
  {
    Console.WriteLine("{0} + {1} * i", Real, Imaginaer);
  }
}
```

Code 34: Erzeugung von Objekten

Wenn Sie das Programm kompilieren und ausführen, erzeugt es folgende Bildschirmausgabe:

```
5 + 3.7 * i
-10.3 + 3.2 * i
```

In der Main-Methode werden mittels

```
CKomplexeZahl ErsteZahl = new CKomplexeZahl();
CKomplexeZahl ZweiteZahl = new CKomplexeZahl();
```

zwei Objekte für komplexe Zahlen angelegt. Zunächst wird dabei also jeweils eine Variable vom Typ `CKomplexeZahl` erzeugt, die eine Referenz auf ein Objekt dieses Typs aufnehmen kann. Dann wird mittels des `new`-Operators ein konkretes Objekt erzeugt, und dessen Referenz in der Variablen gespeichert, so dass nun auf das Objekt zugegriffen werden kann.

Dieser Zugriff kann dabei aber immer nur über Methoden stattfinden, die zu derselben Klasse gehören. Der Aufruf dieser Methoden erfolgt - wie Sie es beispielsweise schon von `Console.WriteLine` kennen - mittels des Punkt-Operators .. Wenn Sie schon ein bisschen in der Dokumentation von C# oder .net gelesen haben, wundern Sie sich vielleicht, dass `Console` kein Objekt ist, sondern eine `Klasse` und `WriteLine` als Methode dieser Klasse nicht auf ein Objekt, sondern direkt mit der Klasse aufgerufen wird. Diese Technik wird weiter unten im Abschnitt *Statische Datenelemente und Methoden* erläutert.

Zu beachten ist noch, dass mit dem Anlegen eines neuen Objektes zwar jedes Mal Speicherplatz für Variablen reserviert wird (jedes Objekt benötigt schließlich seine eigenen Variablen), die Methoden aber - da sie für alle Objekte einer Klasse identisch sind - platzsparend nur einmal im Speicher liegen.

Außerdem - da Objekte Referenztypen sind - bewirkt eine Zuweisung eines bestehenden Objekts an eine weitere Variable keinen Kopiervorgang, sondern beide Variablen stellen das selbe Objekt dar. Dieses Problem kennen Sie schon von Strings. Die Reihenfolge, in der die Klassen in einer Datei angeordnet werden, spielt in C# übrigens keine Rolle.

14.4 Konstruktoren

Die meisten Programme, die mit Objekten arbeiten, gehen so vor wie das Programm im letzten Beispiel: Zuerst wird ein Objekt erzeugt und gleich darauf mit Daten gefüllt. Um nicht immer zwei Aufrufe zu benötigen, kann man beide Schritte mittels einer speziellen Methode in einen zusammenfassen, nämlich mittels des Konstruktors.

Ein Konstruktor ist eine Methode, die automatisch beim Anlegen eines neuen Objekts aufgerufen wird und hauptsächlich zu Initialisierungszwecken dient. Ein Konstruktor trägt dabei immer den Namen der Klasse, zu der er gehört, aber er besitzt als einzige in C# existierende Funktion keinen Rückgabewert (also nicht `void` als Rückgabewert, sondern gar keinen!).

Das folgende Beispiel ist eine Erweiterung der Klasse *CKomplexeZahl* um einen Konstruktor, der neue komplexe Zahlen automatisch auf Null (0 + 0 * i) initialisiert.

```
public class CKomplexeZahl
{
  private double Real;
  private double Imaginaer;

  public CKomplexeZahl()
  {
    Real = 0;
    Imaginaer = 0;
  }

  public void SetReal(double a)
  {
    Real = a;
  }

  public void SetImaginaer(double b)
  {
    Imaginaer = b;
  }

  public double GetReal()
  {
    return Real;
  }

  public double GetImaginaer()
  {
    return Imaginaer;
  }

  public void WriteLine()
  {
    Console.WriteLine("{0} + {1} * i", Real, Imaginaer);
  }
}
```

Code 35: Die Klasse CKomplexeZahl mit Konstruktor

Legt man nun ein neues Objekt für eine komplexe Zahl mittels

```
CKomplexeZahl EineZahl = new CKomplexeZahl();
```

an, so wird automatisch der Konstruktor aufgerufen und weist den Datenelementen *Real* und *Imaginaer* die Werte 0 zu.

Trotzdem hat dieser Konstruktor noch einen Nachteil, denn es wäre praktisch, ihm gleich Werte übergeben zu können. Dies ist auch möglich, indem man den Konstruktor wie eine ganz normale Funktion mit Parametern ausstattet.

```
public CKomplexeZahl(double a, double b)
{
  Real = a;
  Imaginaer = b;
}
```

Nun kann man beispielsweise mittels

```
CKomplexeZahl EineZahl = new CKomplexeZahl(5, 3);
```

die komplexe Zahl 5 + 3 * i erzeugen. Um nun die komplexe Zahl Null zu erzeugen, müsste man den Konstruktor mit zwei Nullen als Parameter aufrufen.

14.5 Überladen von Methoden

Praktischer wäre jedoch, wenn der Konstruktor je nachdem, ob ihm Parameter übergeben werden, die komplexe Zahl mit den entsprechenden Werten initialisiert oder mit Null. Dazu kann man zwei Konstruktoren implementieren, die sich in der Zahl der Parameter (oder, bei gleicher Anzahl der Parameter, in deren Typ) unterscheiden.

An Hand der übergebenen Parameter erkennt C# dann, welcher Konstruktor aufgerufen werden soll. Diese Technik wird als *Überladen* oder auch als *Polymorphimus* bezeichnet und funktioniert in C# nicht nur mit Konstruktoren, sondern auch mit allen anderen Methoden.

Das folgende Programmfragment ist die Erweiterung der Klasse CKomplexeZahl um einen zweiten Konstruktor mit Parametern, der neue komplexe Zahlen auf die übergebenen Werte initialisiert.

```
public class CKomplexeZahl
{
  private double Real;
  private double Imaginaer;

  public CKomplexeZahl()
  {
    Real = 0;
    Imaginaer = 0;
  }

  public CKomplexeZahl(double a, double b)
  {
    Real = a;
    Imaginaer = b;
  }

  public void SetReal(double a)
  {
    Real = a;
  }

  public void SetImaginaer(double b)
  {
    Imaginaer = b;
  }

  public double GetReal()
  {
    return Real;
  }

  public double GetImaginaer()
  {
    return Imaginaer;
  }

  public void WriteLine()
  {
    Console.WriteLine("{0} + {1} * i", Real, Imaginaer);
  }
}
```

Code 36: Die Klasse CKomplexeZahl mit einem zweiten Konstruktor

In der Main-Methode können neue komplexe Zahlen nun entweder mittels

```
CKomplexeZahl EineZahl = new CKomplexeZahl();    // 0 + 0
* i
```

oder

```
CKomplexeZahl EineZahl = new CKomplexeZahl(5, 3);    // 5
+ 3 * i
```

angelegt werden.

14.6 Der this-Zeiger

Wenn Sie sich die beiden Konstruktoren aus dem letzten Beispiel noch einmal genauer ansehen, werden Sie vielleicht zwei kleine Nachteile bemerken: Zum einen heißen die Parameter lediglich *a* und *b* und nicht *Real* und *Imaginaer* (was wesentlich aussagekräftiger wäre), zum anderen ähneln sich die beiden Konstruktoren extrem und so müssen bei Änderungen wahrscheinlich beide Methoden verändert werden, was eine zusätzliche Fehlerquelle darstellt. Daher wäre es praktisch, den gemeinsamen Code nur einmal unterbringen zu müssen.

Das erste Problem, nämlich dem Konstruktor zwei Variablen namens *Real* und *Imaginaer* zu übergeben, scheitert daran, dass der Compiler dann nicht weiß, welche Variable gemeint ist - das Datenelement der Klasse oder der übergebene Parameter.

Um die Parameter trotzdem wie Datenelemente benennen zu können, gibt es den `this`-Zeiger. Wendet man ihn auf eine Variable an, so verwendet C# das Datenelement der Klasse, andernfalls den gleichnamigen Parameter.

Der Konstruktor müsste also folgendermaßen aussehen:

```
public CKomplexeZahl(double Real, double Imaginaer)
{
  this.Real = Real;
  this.Imaginaer = Imaginaer;
}
```

Damit weiß der Compiler, was gemeint ist und kann die Methode problemlos übersetzen.

Auch das zweite Problem, nämlich den Code der Konstruktoren zu vereinfachen, lässt sich mit Hilfe des `this`-Zeigers lösen. Ruft man einen Konstruktor nämlich auf und hängt ihm - getrennt durch `:` - das Schlüsselwort `this` samt eventuellen Parametern an, so wird ein weiterer Konstruktor der Klasse aufgerufen (welcher Konstruktor aufgerufen wird, hängt dann von den Parametern ab).

Da der Aufruf des Konstruktors ohne Parameter dem Aufruf des anderen mit zwei Nullen als Parameter entspricht, kann man ihn entsprechend umschreiben und erhält die folgenden beiden Konstruktoren.

```
public CKomplexeZahl() : this(0, 0)
{
}

public CKomplexeZahl(double Real, double Imaginaer)
{
  this.Real = Real;
  this.Imaginaer = Imaginaer;
}
```

Dieser Code erfüllt den gleichen Zweck wie vorher, aber bei einer Änderung am Code muss nur noch einer der beiden Konstruktoren geändert werden. Das Programm ist also leichter zu überblicken und besser wartbar.

Anzumerken ist noch, dass der über `this` aufgerufene Konstruktor in jedem Fall vor dem aufrufenden Konstruktor ausgeführt wird.

14.7 Objekte als Parameter

Als nächstes werden Sie in die Klasse *CKomplexeZahl* eine Methode einfügen, um zwei komplexe Zahlen zu addieren. Mathematisch gesehen ist die Addition sehr einfach, es werden nämlich einfach jeweils die beiden Real- sowie die beiden Imaginärteile addiert.

Ohne OOP hätten Sie eine Funktion geschrieben, der zwei komplexe Zahlen (wie auch immer) übergeben werden und die diese dann verarbeitet. Mit der OOP haben Sie zwei Möglichkeiten, da der Methodenaufruf ja in jedem Fall an ein Objekt gekoppelt ist.

Entweder benutzen Sie nach wie vor eine Methode mit zwei Parametern. Die beiden Parameter sind dann die zu addierenden Zahlen, das Objekt, über das die Methode aufgerufen wird, speichert dann das Ergebnis.

Oder Sie benutzen eine Methode mit nur einem Parameter und das aufrufende Objekt stellt diesmal außer dem Objekt, in dem das Ergebnis gespeichert werden soll, auch den zweiten Summanden dar.

Dank Polymorphismus kann man der Klasse sogar beide Methoden mit dem gleichen Namen (beispielsweise *Addieren*) hinzufügen, da sie sich durch die Anzahl der Parameter unterscheiden lassen. Die erweiterte Klasse sieht also folgendermaßen aus (s. Folgeseite):

```
public class CKomplexeZahl
{
  private double Real;
  private double Imaginaer;

  public CKomplexeZahl() : this(0, 0)
  {
  }

  public CKomplexeZahl(double Real, double Imaginaer)
  {
    this.Real = Real;
    this.Imaginaer = Imaginaer;
  }

  public void SetReal(double a)
  {
    Real = a;
  }

  public void SetImaginaer(double b)
  {
    Imaginaer = b;
  }

  public double GetReal()
  {
    return Real;
  }

  public double GetImaginaer()
  {
    return Imaginaer;
  }

  public void Addieren(CKomplexeZahl EineZahl)
  {
    Real += EineZahl.Real;
    Imaginaer += EineZahl.Imaginaer;
  }

  public void Addieren(CKomplexeZahl ErsteZahl, CKomplexeZahl
  ZweiteZahl)
  {
    Real = ErsteZahl.Real + ZweiteZahl.Real;
    Imaginaer = ErsteZahl.Imaginaer + ZweiteZahl.Imaginaer;
  }

  public void WriteLine()
  {
    Console.WriteLine("{0} + {1} * i", Real, Imaginaer);
  }
}
```

Code 37: Die Klasse CKomplexeZahl um Polymorphismus erweitert

Interessant an den beiden Methoden ist, dass sie ohne weiteres auf die Datenelemente der als Parameter übergebenen Objekte zugreifen können, obwohl diese als `private` definiert sind. Dies funktioniert deshalb, da das als Parameter übergebene Objekt und das, über welches die Methode aufgerufen wird, aus der selben Klasse stammen. Würde man ein Objekt eines anderen Typs übergeben, so müsste man statt dessen mit entsprechenden Methodenaufrufen arbeiten.

14.8 Statische Datenelemente und Methoden

Inzwischen kennen Sie die wichtigsten Grundlagen der objektorientierten Programmierung und wissen, was Klassen und Objekte sind. Trotzdem fehlt Ihnen noch ein Element, um die bisher geschriebenen Programme vollständig zu verstehen. Schauen Sie sich noch einmal die Zeile an, in der die Main-Methode definiert wird.

```
public static void Main()
```

Warum die Main-Methode `public` sein muss, ist inzwischen klar, sonst könnte man das Programm nämlich gar nicht aufrufen. Was Sie aber noch nicht kennen, ist das Schlüsselwort `static`.

Die Angabe `static` kann sowohl bei Datenelementen als auch bei Methoden gemacht werden und bewirkt, dass immer genau eine Instanz dieses Datenelementes oder dieser Methode existiert, unabhängig davon, wie viele Objekte existieren (und, ob überhaupt eines existiert).

Statische Datenelemente können beispielsweise dazu verwendet werden, die Anzahl der von einer Klasse erzeugten Objekte zu zählen, indem die Variable beim Anlegen eines neuen Objektes durch den Konstruktor automatisch um eins erhöht wird. Dadurch, dass das Datenelement in jedem Fall nur genau einmal existiert, ist sichergestellt, dass alle Objekte auf das selbe Datenelement zugreifen.

```
public class CZaehler
{
  private static AnzahlObjekte = 0;

  public CZaehler()
  {
    Console.WriteLine("Das Objekt Nummer {0} wurde gerade
erstellt.", AnzahlObjekte++);
  }
}
```

Code 38: Statische Datenelemente

Werden in einem Programm nun beispielsweise vier Objekte der Klasse *CZaehler* angelegt, so gibt jedes Objekt bei seiner Erstellung auf dem Bildschirm aus, das wie vielte dieser Klasse es ist. Das Programmfragment

```
ErstesObjekt = new CZaehler();
ZweitesObjekt = new CZaehler();
DrittesObjekt = new CZaehler();
ViertesObjekt = new CZaehler();
```

gibt beispielsweise

```
Das Objekt Nummer 0 wurde gerade erstellt.
Das Objekt Nummer 1 wurde gerade erstellt.
Das Objekt Nummer 2 wurde gerade erstellt.
Das Objekt Nummer 3 wurde gerade erstellt.
```

auf dem Bildschirm aus. Statische Datenelemente kommen in der Praxis nicht sehr oft vor, aber falls einmal eine entsprechende Funktionalität benötigt wird - nämlich, dass alle Objekte gemeinsam auf einen gemeinsamen Wert zugreifen - , so bieten sie eine sehr elegante Lösung.

Statische Methoden hingegen kommen in der Praxis etwas öfter vor. Ihr Sinn besteht darin, eine Methode auch aufrufen zu können, ohne vorher ein Objekt der entsprechenden Klasse anlegen zu müssen. Deswegen ist beispielsweise die Main-Methode statisch, da von der Klasse, in der Sie enthalten ist, nie ein Objekt angelegt wird. Damit die Funktion trotzdem existiert und aufrufbar ist, muss sie als statisch definiert werden.

Wie bei den Objekten haben Sie auch schon mit weiteren statischen Methoden gearbeitet, ohne es zu wissen. Beispielsweise ist der Befehl für die Bildschirmausgabe, `WriteLine`, eine statische Methode der Klasse `Console`, wie oben bereits erwähnt wurde: Sie legen nämlich nie ein Objekt der Klasse `Console` an, sondern verwenden die Klasse immer so, als sei sie bereits ein Objekt. Auch diese Funktionalität wird durch das Schlüsselwort `static` ermöglicht.

14.9 Ausblick

Im nächsten Kapitel werden Sie lernen, was Vererbung ist und wie man sie in C# einsetzt.

15 Vererbung

In diesem Kapitel werden Sie lernen, was Vererbung ist und wie man sie in C# einsetzt.

Bevor Sie sich mit diesem Kapitel beschäftigen, sollten Sie sicher sein, dass Sie das Grundprinzip der objektorientierten Programmierung verstanden haben und wissen, was Klassen und Objekte sind.

Dieses Kapitel enthält im Gegensatz zu den vorangegangenen keine Programmbeispiele, da die besprochenen Techniken erst bei etwas größeren Projekten sinnvoll einsetzbar sind. Da es sich aber um Grundlagen handelt, werden sie hier dennoch besprochen. Angewendet werden sie dann im zweiten Teil des Guide, in dem es unter anderem um die Programmierung grafischer Oberflächen geht.

15.1 Vererbung

Im letzten Kapitel haben Sie gelernt, wie man Klassen erstellt und dann Objekte von diesen Klassen erzeugt. Allerdings haben Sie alle Klassen immer von Grund auf neu programmiert. Dieses Vorgehen ist in der Praxis allerdings oft nicht notwendig.

Häufig gibt es nämlich bereits eine Klasse, die eine ähnliche - vielleicht etwas allgemeinere - Aufgabe erfüllt. Statt deren Funktionalität nachzuprogrammieren, können Sie auf die vorhandene Klasse zugreifen und sie erweitern. Ihre eigene, neu erstellte Klasse enthält dann neben den von Ihnen entwickelten Methoden und Datenelemente auch alle aus der vorhandenen Klasse.

Als einfaches und anschauliches Beispiel können Sie sich eine Klasse vorstellen, die Fenster auf dem Bildschirm darstellt. Eine solche Klasse könnte beispielsweise Datenelemente für die Beschriftung, die Position und den Inhalt des Fensters beinhalten, sowie Methoden, um diese Datenelemente zu verändern oder auszulesen.

Wenn Sie nun eine Klasse für ein Fenster benötigen, welches standardmäßig einen Rollbalken enthält, müssen Sie nicht die komplette Fensterfunktionalität neu programmieren, sondern erstellen sozusagen eine Kopie der Fensterklasse, erweitern diese Kopie um die gewünschten Eigenschaften und geben dieser nun umfangreicheren Version einen neuen Namen. Diese Technik wird in der OOP als *Vererbung* oder *Ableitung* bezeichnet.

Die Klasse, von der vererbt wird, heißt dabei *Basisklasse* (manchmal wird sie auch als *Superklasse* bezeichnet), während die Klasse, die erbt, *abgeleitete Klasse* genannt wird. Sie *erweitert* die ursprüngliche Funktionalität der Basisklasse.

Vererbung wird in Applikationen, die mit C# entwickelt werden, sehr oft eingesetzt, da man so von den vorgefertigten und sehr leistungsfähigen Klassen des .net Framework profitieren kann. Eine solch umfangreiche Ansammlung von Klassen zur Applikationsentwicklung wie das .net Framework wird auch als *Klassenbibliothek* bezeichnet. Andere bekannte Klassenbibliotheken zur Anwendungsentwicklung sind beispielsweise die MFC unter Win32 oder QT unter Linux.

Ein Vorteil der OOP, der beispielsweise die Nutzung von Klassenbibliotheken erst ermöglicht, ist, dass Sie bei der Vererbung den Code der Basisklasse nicht zu kennen brauchen. Die Basisklasse stellt für Sie lediglich eine Blackbox dar, die eine bestimmte Aufgabenstellung erledigt. Sie müssen Sie also nur ableiten und die Funktionen hinzufügen, die Sie zusätzlich benötigen.

Eine neue Klasse von einer bestehenden abzuleiten, ist in C# sehr einfach. Dazu muss nämlich bei der Klassendefinition hinter dem Namen der neuen Klasse lediglich ein Doppelpunkt sowie der Name der abzuleitenden Klasse angegeben werden. Um bei dem Beispiel mit den Fenstern zu bleiben, könnte eine Definition der Klasse für Fenster mit Rollbalken also folgendermaßen aussehen.

```
public class CFensterRollbalken : CFenster
{
}
```

Damit haben Sie bereits eine neue Klasse namens *CFensterRollbalken* erzeugt, die genau die selbe Funktionalität wie die Basisklasse *CFenster* hat. Zusätzlich können Sie aber - und zwar ohne auch nur eine Zeile Code der Klasse *CFenster* zu kennen - eigene Methoden und Datenelemente hinzufügen, um Ihre neue Klasse leistungsfähiger zu machen.

Damit aber immer noch aus der neuen, abgeleiteten Klasse heraus auf die Methoden und Datenelemente der Basisklasse explizit zugegriffen werden kann, existiert das Schlüsselwort base. Durch base wird Ihnen sozusagen eine Referenz auf die Basisklasse bereitgestellt.

Selbst, wenn Sie eine Klasse nicht ableiten, existiert immer eine Basisklasse, von der automatisch alle anderen Klassen abgeleitet werden. Auch die im .net Framework vorgerfertigten Klassen sind von dieser eine Basisklasse abgeleitet. Diese Klasse heißt `System.Object` und implementiert lediglich die für jede Klasse absolut notwendigen Grundfunktionen.

Zu diesen Grundfunktionen gehört beispielsweise auch eine Methode namens `ToString`, die Sie eigentlich auch in jeder Ihrer eigenen Klassen implementieren sollten. `ToString` dient dazu, den Inhalt eines Objekts in einen String umzuwandeln, so dass er in sinnvoller Form ausgegeben oder weiter verarbeitet werden kann.

Falls Sie bereits Programmiererfahrung in C++ besitzen, haben Sie vielleicht schon von Mehrfachvererbung gehört. Bei der Mehrfachvererbung besitzt eine Klasse nicht nur eine, sondern gleich mehrere Basisklassen. Diese leistungsfähige Funktion wurde unter C++ neu eingeführt, leider war sie aber auch eine der größten Fehlerquellen. Da Mehrfachvererbung trotz seiner Leistungsfähigkeit äußerst selten eingesetzt wird, bietet C# diese Funktion nicht mehr an, es existiert also nur noch die Einfachvererbung.

Falls Sie - aus welchen Gründen auch immer - dennoch einmal Mehrfachvererbung benötigen sollten (was allerdings sehr unwahrscheinlich ist), steht Ihnen diese Möglichkeit trotzdem zur Verfügung, nämlich über sogenannte Interfaces, die weiter unten noch vorgestellt werden.

15.2 Zugriffsmodifizierer

Bisher haben Sie zum Definieren der Zugriffsmöglichkeiten auf Datenelemente, Methoden und Klassen nur zwei Modifizierer benutzt, nämlich `public` und `private`. Während der erste das zugehörige Element für jeglichen Zugriff freigibt, erlaubt zweiterer Zugriff nur noch für Methoden aus der selben Klasse.

Bisher kamen Sie mit diesen beiden Schlüsselwörtern aus. Sobald Sie aber Vererbung beherrschen und eine Klasse ableiten möchten, werden Sie vielleicht feststellen, dass C# es mit dem Zugriffsschutz genauer nimmt, als Sie angenommen haben. Sie haben nämlich auf ein Element, welches als `private` definiert ist, nicht nur von einer gänzlich anderen Klasse keinen Zugriff, sondern auch in der abgeleiteten Klasse nicht. Da diese Art des Zugriffs manchmal aber wichtig sein kann, das Element andererseits aber auch nicht als `public` definiert werden soll, gibt es noch weitere Modifizierer, die genau diese Problematik lösen.

Modifizierer	*innerhalb der Klasse*	*innerhalb der Ableitungen*	*außerhalb der Klasse*	*außerhalb der Anwendung*
public	ja	ja	ja	ja
internal	ja	ja	ja	nein
protected	ja	ja	nein	nein
private	ja	nein	nein	nein

Tabelle 11: Zugriffsmodifizierer

Der Modifizierer `protected` bietet also den Schutz von `private`, dehnt die Zugriffsmöglichkeit aber auch noch auf abgeleitete Klassen aus. Um ein Element innerhalb der ganzen Anwendung freizugeben, es aber vor anderen Anwendungen zu schützen, gibt es in C# den neuen Modifizierer `internal`. Wenn Sie ein Element als `internal` kennzeichnen, ist es innerhalb der ganzen Anwendung `public`, aber für alle anderen Anwendungen `private`.

Dennoch werden Sie in den meisten Fällen mit den Schlüsselwörtern `public` und `private` hinkommen, `protected` und vor allem `internal` werden relativ selten gebraucht. Trotzdem lohnt es sich, beide im Sinn zu behalten, da ihre Verwendung manchmal die einzige Lösung für eine Problem darstellt.

15.3 Überschreiben von Methoden

Sie wissen jetzt, wie Sie Klassen vererben, aber es gibt noch einen Ausnahmefall, der noch nicht besprochen wurde. Stellen Sie sich vor, Sie möchten in einer abgeleiteten Klasse eine Methode implementieren, aber in der Basisklasse existiert eine Methode mit diesem Namen bereits. Nun stellt sich für den Compiler die Frage, welche Methode er aufrufen soll, die aus der Basisklasse oder die neue.

Dazu gibt es zwei Lösungsmöglichkeiten. Die erste ist die, dass Sie wirklich die Methode der Basisklasse durch Ihre eigene, neue Methode ersetzen. Damit dies aber nicht versehentlich geschehen kann, muss die Methode in der Basisklasse als überschreibbar gekennzeichnet sein. Dies geschieht mit dem Schlüsselwort `virtual`. Außerdem muss auch bei der neuen Methode durch das Schlüsselwort `override` explizit angegeben werden, dass sie die Basisversion ersetzen soll.

Bitte beachten Sie, dass - auch wenn Sie die Methode nun überschreiben können - die Zugriffsmöglichkeiten (`public`, ...) nicht mehr verändert werden können. Daher darf bei der Definition der neuen Version der Methode auch kein Modifizierer angegeben werden, statt dessen ersetzt das Schlüsselwort `override` den sonst üblichen Modifizierer.

Außerdem muss man darauf achten, dass durch das Überschreiben einer Methode durch `virtual` und `override` nicht nur die Methode der abgeleiteten Klasse ersetzt wird, sondern auch die ursprüngliche Methode der Basisklasse! So wird beispielsweise nach einer expliziten Typkonvertierung eines Objekts der abgeleiteten Klasse in ein Objekt der Basisklasse immer noch die neue Methode aufgerufen. Durch die Angabe von `virtual` und `override` wird sozusagen sichergestellt, dass immer die neueste Version einer Methode aufgerufen wird.

Die zweite Möglichkeit, ist das Verbergen der ursprünglichen Methode in der abgeleiteten Klasse. Dann weiß diese nichts davon, dass bereits eine Methode gleichen Namens in der Basisklasse existiert. Um eine Methode zu verbergen, muss in der abgeleiteten Klasse das Schlüsselwort `new` verwendet wird.

Dies funktioniert im Gegensatz zu `override` auch dann, wenn die Methode der Basisklasse nicht als `virtual` definiert ist. Außerdem ersetzt `new` - im Gegensatz zu `override` - nicht den Modifizierer, sondern es muss zusätzlich zu diesem gesetzt werden. Durch Verwendung von `new` ist die neue Version der Methode also nur in der abgeleiteten Klasse gültig, in der Basisklasse hingegen gilt immer noch die ursprüngliche Version.

15.4 Versiegelte und abstrakte Klassen

Die Zugriffsmodifizierer, die Sie bisher kennengelernt haben, konnten nicht nur für Datenelemente und Methoden eingesetzt werden, sondern auch für Klasse (wenn auch zugegebenermaßen eine Klasse, die als `private` definiert ist, relativ wenig Sinn macht).

Es gibt aber außer diesen Modifizierern noch zwei weitere, die allerdings nur für Klassen gültig sind, nämlich `sealed` und `abstract`.

Wenn Sie eine Klasse mit dem Schlüsselwort `sealed` kennzeichnen, so ist sie *versiegelt* und kann nicht mehr als Basisklasse für weitere Klassen verwendet werden, sie kann also nicht mehr abgeleitet werden. Auch wenn dies auf den ersten Blick sehr nützlich bei der (kommerziellen) Weitergabe von Klasse zu sein scheint, so widerspricht es doch gänzlich dem Gedanken der OOP. Deshalb sollten Sie `sealed` nur äußerst sparsam einsetzen.

Denkbar wäre beispielsweise eine Klasse, die nur für interne Debug-Zwecke verwendet wird und daher nicht von allgemeinem Interesse ist. Um eine unkontrollierte Verwendung auszuschließen, könnte man eine solche Klasse als `sealed` kennzeichnen.

Außerdem kann der Einsatz von `sealed` sinnvoll bei Klassen sein, die eine bestimmte Funktionalität implementieren, die verändert oder erweitert keinen Sinn ergibt. Im .net Framework existieren einige solche Klassen, beispielsweise `Math`, die Sie bereits kennengelernt haben.

Mit dem Schlüsselwort `abstract` hingegen wird eine Klasse gekennzeichnet, die nur aus Deklarationen von Methoden besteht - also keinen ausführbaren Code enthält - und außerdem auch keine Datenelemente enthält.

Alle Methoden einer abstrakten Klasse sind dann natürlich implizit `virtual` und müssen, sobald die Klasse abgeleitet wird, mittels des Schlüsselworts `override` überschrieben und implementiert werden. Von abstrakten Klassen können keine Objekte angelegt werden, da diese weder Datenelemente noch ausführbare Methoden enthalten würden.

Beachten Sie bitte, dass sich die Schlüsselwörter `sealed` und `abstract` gegenseitig ausschließen, da eine abstrakte Klasse ableitbar sein muss, was `sealed` gerade verhindern würde.

15.5 Interfaces

Interfaces, die im Deutschen auch oft als *Schnittstellen* bezeichnet werden, haben eine ähnliche Aufgabe wie abstrakte Klassen. Sie implementieren weder Datenelemente noch ausführbaren Code, sondern lediglich Deklarationen von Methoden. Auch sie müssen vererbt und in der abgeleiteten Klasse implementiert werden.

Der Unterschied zu abstrakten Klassen ist, wie oben bereits angedeutet wurde, dass eine Klasse lediglich eine Basisklasse haben kann, aber beliebig viele Interfaces implementieren kann. Interfaces dienen also zur Mehrfachvererbung.

Falls eine Klasse neben einer Basisklasse noch von mindestens einem Interface abgeleitet ist, so muss bei der Klassendeklaration die Basisklasse vor den Interfaces angegeben werden.

Die Syntax eines Interfaces unterscheidet sich von der einer abstrakten Klasse nur dadurch, dass das Schlüsselwort `class` durch `interface` ersetzt ist und das Schlüsselwort `abstract` entfällt, da Interfaces per Definition abstrakt sind.

Folgendes Programmfragment stellt ein Interface für komplexe Zahlen dar, welches von der Klasse *CKomplexeZahl*, die Sie im letzten Kapitel entwickelt haben, implementiert werden könnte.

```
interface IKomplexeZahl
{
  void SetReal(double Real);
  void SetImaginaer(double Imaginaer);
  double GetReal();
  double GetImaginaer();
  void WriteLine();
}
```

Um dieses Interface tatsächlich in der Klasse *CKomplexeZahl* zu implementieren, müsste die Zeile

```
public class CKomplexeZahl
```

in

```
public class CKomplexeZahl : IKomplexeZahl
```

geändert werden. Interfaces können also auch dazu dienen, erst einmal grobe Anforderungen an die Methoden einer Klasse zu beschreiben, welche dann in der Klasse implementiert werden müssen.

15.6 Ausblick

Im nächsten Kapitel werden Sie erweiterte Klassenkonzepte von C# kennen lernen, nämlich Eigenschaften, Indexer, Delegates, Events und das Überladen von Operatoren.

16 Klassenkonzepte

Im diesem Kapitel werden Sie erweiterte Klassenkonzepte von C# kennen lernen, nämlich Eigenschaften, Indexer, Delegates, Events und das Überladen von Operatoren.

All diese Konzepte sind sehr leistungsfähige Techniken zur effizienten Arbeit mit Klassen, die derzeit - abgesehen vom Überladen von Operatoren - in keiner anderen Programmiersprache verfügbar sind.

16.1 Eigenschaften

Oft kommt es vor, dass Datenelemente zwar `private` sein sollen (beispielsweise damit sie nicht beliebig verändert werden können, sondern nur durch Methoden, die eine Wertevalidierung durchführen), aber außer der Set- und Get-Methode kaum eine oder vielleicht auch gar keine Methode für diese Datenelemente aufgerufen wird. Um im Programm nicht jedes Mal die Zugriffsmethoden aufrufen zu müssen, kann man den Wert statt als Datenelement auch als Eigenschaft der Klasse speichern.

Eigenschaften sind sozusagen Dummy-Datenelemente, die sich wie als `public` definierte Datenelemente verhalten. Intern wird aber bei jedem Zugriff eine Zugriffsmethode aufgerufen, die den Wert in einem als `private` definierten Datenelement speichert. Die beiden Zugriffsfunktionen der Eigenschaft sind ebenfalls öffentlich (`public`) und heißen *get* beziehungsweise *set*.

Man könnte also die Datenelemente der Klasse *CKomplexeZahl* auch folgendermaßen als Eigenschaften implementieren (s. Folgeseite).

```
public class CKomplexeZahl
{
  private double InternReal;
  public double Real
  {
    get
    {
      return InternReal;
    }
    set
    {
      InternReal = value;
    }
  }
  private double InternImaginaer;
  public double Imaginaer
  {
    get
    {
      return InternImaginaer;
    }
    set
    {
      InternImaginaer = value;
    }
  }
  public CKomplexeZahl() : this(0, 0)
  {
  }

  public CKomplexeZahl(double Real, double Imaginaer)
  {
    InternReal = Real;
    InternImaginaer = Imaginaer;
  }

  public void Addieren(CKomplexeZahl EineZahl)
  {
    InternReal += EineZahl.InternReal;
    InternImaginaer += EineZahl.InternImaginaer;
  }

  public void Addieren(CKomplexeZahl ErsteZahl, CKomplexeZahl
  ZweiteZahl)
  {
    InternReal = ErsteZahl.InternReal + ZweiteZahl.InternReal;
    InternImaginaer = ErsteZahl.InternImaginaer +
    ZweiteZahl.InternImaginaer;
  }

  public void WriteLine()
  {
    Console.WriteLine("{0} + {1} * i", Real, Imaginaer);
  }
}
```

Code 39: Datenelemente als Eigenschaften implementieren

Nun kann man nach wie vor nicht direkt auf die Datenelemente *InternReal* und *InternImaginaer* zugreifen, aber durch die öffentlichen Zugriffsfunktionen der öffentlichen Eigenschaften *Real* und *Imaginaer* kann man indirekt und ohne Methodenaufruf auf sie zugreifen. Folgende Anweisungen sind also gültig:

```
CKomplexeZahl EineZahl = new CKomplexeZahl();
EineZahl.Real = 2.5;
EineZahl.Imaginaer = 3;
EineZahl.WriteLine();   // 2.5 + 3 * i
```

Ebenso könnte lesend auf *EineZahl.Real* und *EineZahl.Imaginaer* zugegriffen werden. Bitte beachten Sie, dass die in der set-Funktion verwendete Variable *value* fest vorgegeben ist und den Wert repräsentiert, mit der die Funktion aufgerufen wird.

Außer der Möglichkeit, komfortabel auf private Variablen zuzugreifen, haben Eigenschaften noch einen weiteren Vorteil: Man kann mit ihnen den lesenden oder schreibenden Zugriff auf Datenelemente verbieten, indem nur eine der beiden Zugriffsfunktionen implementiert wird.

get- und set-Funktionen		
	get-Funktion	set-Funktion
implementiert	lesender Zugriff erlaubt	schreibender Zugriff erlaubt
nicht implementiert	lesender Zugriff verboten	schreibender Zugriff verboten

Tabelle 12: get- und set-Funktionen

16.2 Indexer

Wie Sie bereits wissen, können Sie bei Strings auf einzelne Zeichen lesend zugreifen, wenn Sie einen String als Array betrachten und die zu lesende Position in eckigen Klammern übergeben.

Folgendes Beispiel veranschaulicht diese Technik noch einmal.

```
using System;

public class Text
{
  public static void Main()
  {
    string Name = "Microsoft C#";
    Console.WriteLine(Name[0]);   // M
  }
}
```

Code 40: Einzelnde Zeichen eines Strings einlesen

Ein ähnliches Verhalten können Sie in C# auch für eigene Klassen entwickeln, nämlich über sogenannte *Indexer*. Indexer sind dabei Methoden, die - ähnlich wie Eigenschaften - mittels `get` und `set` auf Datenelemente zugreifen können und dabei einen ihnen übergebenen Index berücksichtigen.

Als Beispiel soll folgende Klasse dienen, die IP-Adressen speichern kann. Mittels des Indexers kann man auf einzelne Elemente der IP-Adresse lesend oder schreibend zugreifen, wie die `Main`-Methode zeigt.

```
using System;

public class CIPAdresse
{
  private int[] IPFeld = new int[4];

  public CIPAdresse(int Feld1, int Feld2, int Feld3, int
  Feld4)
  {
    IPFeld[0] = Feld1;
    IPFeld[1] = Feld2;
    IPFeld[2] = Feld3;
    IPFeld[3] = Feld4;
  }

  public CIPAdresse(int[] IPAdresse)
  {
    for(int i = 0; i < 4; i++)
      IPFeld[i] = IPAdresse[i];
  }

  public int this[int Index]
  {
    get
    {
      return IPFeld[Index];
    }
    set
    {
      IPFeld[Index] = value;
    }
  }
}

public class Test
{
  public static void Main()
  {
    CIPAdresse myIP = new CIPAdresse(192, 168, 0, 1);
    Console.WriteLine("Die IP-Adresse lautet:
    {0}.{1}.{2}.{3}", myIP[0], myIP[1], myIP[2], myIP[3]);
    // ergibt 196.168.0.1
    myIP[3] = 2;
    Console.WriteLine("Die IP-Adresse lautet:
    {0}.{1}.{2}.{3}", myIP[0], myIP[1], myIP[2], myIP[3]);
    // ergibt 196.168.0.2
  }
}
```

Code 41: Speichern von IP-Adressen

Wie man sieht, unterscheiden sich Indexer von Eigenschaften nur dadurch, dass Indexer keinen eigenen Namen haben, sondern einfach nur `this` heißen. Außerdem besitzen sie eine Parameterliste, die allerdings im Gegensatz zu Funktionen nicht in runden, sondern in eckigen Klammern übergeben wird.

Diese Parameterliste ist nicht auf einen Parameter beschränkt, sondern kann auch mehrere Parameter aufnehmen. Außerdem sind die übergebenen Parameter nicht auf `int` als Typ festgelegt, sondern es kann jeder beliebige Typ verwendet werden. Allerdings dürfen die Parameter weder mit dem Schlüsselwort `ref` noch mit `out` gekennzeichnet werden.

Wie bei Eigenschaften ist es auch bei Indexern möglich, nur die Methode `get` oder `set` zu implementieren, um nur den lesenden oder schreibenden Zugriff zu ermöglichen.

16.3 Delegates

Manchmal ist es notwendig oder nützlich, einer Methode eine andere Methode als Parameter übergeben zu können. Dazu verwendet man bei C# sogenannte *Delegates*, die eine Referenz auf eine Methode aufnehmen können.

Delegates ähneln den Funktionszeigern in C++, sind aber typsicher. Das heißt, Delegates können nur Methoden referenzieren, die in ihrer Parameteranzahl und in ihrem Rückgabewert exakt der Deklaration des Delegates entsprechen.

Ein Delegate wird mittels des Schlüsselwortes `delegate` außerhalb einer Klasse deklariert und besitzt folgende Syntax.

```
Modifizierer delegate Rückgabewert DelegateName (Parameter);
```

Delegates können dabei aber sowohl auf statische als auch auf Instanzmethoden verweisen. Das folgende einfache Beispiel zeigt, wie Delegates eingesetzt werden können.

```
using System;

delegate void Methode(string Text);

public class CHilfsKlasse
{
  public void InstanzMethode(string Text)
  {
    Console.WriteLine("Der Parameter der Instanzmethode lautet
\"{0}\".", Text);
  }

  public static void StatischeMethode(string Text)
  {
    Console.WriteLine("Der Parameter der statischen Methode
lautet \"{0}\".", Text);
  }
}

public class CHauptKlasse
{
  public static void Main()
  {
    CHilfsKlasse HilfsKlasse = new CHilfsKlasse();

    // Die Instanzmethode dem Delegate zuweisen
    Methode EinDelegate = new
Methode(HilfsKlasse.InstanzMethode);
    EinDelegate("Hallo!");

    // Die statische Methode dem Delegate zuweisen
    EinDelegate = new Methode(CHilfsKlasse.StatischeMethode);
    EinDelegate("Tschüss!");
  }
}
```

Code 42: Delegates

Dieses Programm gibt folgenden Text auf dem Bildschirm aus.

```
Der Parameter der Instanzmethode lautet "Hallo!".
Der Parameter der statischen Methode lautet "Tschüss!".
```

Vielleicht ist Ihnen aufgefallen, dass bei der Zuweisung einer Methode an den Delegate hinter dem Methodennamen keine runden Klammern angegeben wurden. Das ist notwendig, damit der Compiler zwischen einer Referenz auf eine Methode und einem Methodenaufruf unterscheiden kann. Würde man die runden Klammern angeben, würde C# versuchen, die Methode aufzurufen.

Bitte beachten Sie, dass bei der Zuweisung von Instanzmethoden eine Instanz einer Klasse, bei der Zuweisung einer statischen Methode aber die Klasse selbst vorangestellt werden muss, da sich statische Methoden nicht auf ein einzelnes Objekt, sondern auf die Klasse insgesamt beziehen.

Bemerkenswert an Delegates ist, dass sie einem einzelnen Delegate mehrere Methoden zuweisen können, indem Sie den `+=`-Operator verwenden. Im letzten Beispiel könnten Sie also auch die statische und die Instanzmethode gleichzeitig aufrufen, indem Sie in der `Main`-Methode folgenden Code verwenden.

```
Methode EinDelegate = new
Methode(HilfsKlasse.InstanzMethode);
EinDelegate += new Methode(HilfsKlasse.StatischeMethode);
```

Wenn Sie nun den Delegate mittels

```
Methode("Hallo!");
```

aufrufen, werden intern beide Methoden aufgerufen und es findet die Bildschirmausgabe

```
Der Parameter der Instanzmethode lautet "Hallo!".
Der Parameter der statischen Methode lautet "Hallo!".
```

statt. Genau so, wie Sie mehrere Methoden hinzufügen können, können Sie auch einzelne Methoden wieder aus dem Delegate entfernen, indem Sie den `-=`-Operator verwenden.

Um alle Methoden aus einem Delegate zu entfernen, muss man allerdings nicht jede Methode einzeln entfernen, sondern kann dem Delegate einfach eine `null`-Referenz zuweisen.

16.4 Events

Nachdem Sie nun wissen, was man unter einem Delegate versteht, verfügen Sie über das nötige Grundwissen, um *Events* in C# zu verstehen. Events ermöglichen es, einen Delegate anzugeben, der beim Auftreten eines bestimmten Ereignisses im Code aufgerufen wird.

Anschaulich können Sie sich beispielsweise vorstellen, dass der Mausklick auf eine Schaltfläche ein solches Event auslöst. Durch das Event wird ein Delegate aufgerufen, mit dem wiederum beliebig viele Methoden verknüpft sein können, die auf den Mausklick reagieren sollen.

Bei der Deklaration von Events gibt es zwei Möglichkeiten. Die einfachere Variante verbindet lediglich das Event mit einem Delegate, die komplexere Variante bietet darüber hinaus noch die Möglichkeit, zwei Methoden `add` und `remove` zu implementieren, die zum Hinzufügen und Entfernen von Ereignishandlern verwendet werden.

Wird eine der beiden Methoden implementiert, so muss auch die andere implementiert werden. Allerdings wird auf diese erweiterte Eventdeklaration nicht näher eingegangen, da sie nur äußerst selten benötigt wird.

Die Syntax lautet entweder

```
Modifizierer event Delegate EreignisName
```

oder

```
Modifizierer event Delegate EreignisName
{
  add
  {
    // Anweisungen
  }

  remove
  {
    // Anweisungen
  }
}
```

Um Events verwenden zu können, müssen Sie folgendermaßen vorgehen. Falls Sie ein eigenes Event definieren wollen, müssen Sie zunächst einen Delegate deklarieren, der aufgerufen werden soll. Bei vorgefertigten Events des .net Framework reicht es aus, wenn Ihnen der Name des Delegate bekannt ist.

Als nächstes muss eine Klasse erstellt werden, die das Event definiert. Diese Klasse muss daher eine `event`-Deklaration sowie mindestens eine Methode enthalten, die das Event auslöst. Optional kann noch eine Methode implementiert werden, die überprüft, dass überhaupt eine Instanz des Delegate existiert, der mit dem Event verbunden werden soll. Wird diese Methode weggelassen, muss die Überprüfung beim Auslösen des Events durchgeführt werden.

Außerdem müssen Sie mindestens eine Klasse erstellen, welche die Methoden enthält, die durch den Delegate aufgerufen werden sollen. Des weiteren müssen diese Klassen dafür sorgen, dass die Methoden auch mittels `+=` mit dem Delegate verbunden werden.

Schließlich müssen Sie noch ein Objekt der Klasse erstellen, welche die Eventdefinition enthält, um das Event verwenden zu können. Nun kann das Event ausgelöst werden.

Das folgende Beispiel soll diese doch etwas theoretischen Ausführungen veranschaulichen.

```
using System;

public delegate void Methode();

public class CEventKlasse
{
  public event Methode EinEvent;

  public void EventAusloesen()
  {
    if (EinEvent != null)
      EinEvent();
  }
}
public class CHauptKlasse
{
  private static void Ausgabe()
  {
    Console.WriteLine("Diese Methode wird aufgerufen, sobald
das Event ausgelöst wird.");
  }

  public static void Main()
  {
    CEventKlasse EventKlasse = new CEventKlasse();

    EventKlasse.EinEvent += new Methode(Ausgabe);
    EventKlasse.EventAusloesen();
  }
}
```

Code 43: Events

Vielleicht ist Ihnen aufgefallen, dass die Zuweisung der Methode `Ausgabe` an den Delegate nicht - wie man vielleicht erwartet hätte - mittels des `=`-Operators, sondern mittels des `+=`-Operators erfolgt ist.

Diese Art der Zuweisung ist notwendig, um ein versehentliches Entfernen aller bereits mit dem Delegate verbundenen Methoden zu verhindern.

16.5 Überladen von Operatoren

Unter dem Überladen von Operatoren versteht man das neu definieren von Operatoren für selbst entwickelte Klassen. So wäre es beispielsweise möglich, die Klasse zur Implementierung von komplexen Zahlen um den Operator + zu erweitern, um mit komplexen Zahlen genauso einfach und intuitiv umgehen zu können, wie mit den eingebauten Datentypen.

Außer Berechnungen können auch Vergleichs- und Konvertierungs-operatoren überladen werden. Zuweisungs-, Bedingungs- sowie logische Operatoren können allerdings nicht überladen werden.

Um unäre oder binäre Operatoren zu überladen, wird als Syntax

```
public static Rückgabetyp operator Operator (Operandtyp
OperandName)
{
  // Anweisungen
}
```

für unäre Operatoren und

```
public static Rückgabetyp operator Operator (Operandtyp1
OperandName1, Operandtyp2 OperandName2)
{
  // Anweisungen
}
```

für binäre Operatoren verwendet.

Um einen im- oder expliziten Konvertierungsoperator zu überladen, wird hingegen als Syntax

```
public static implicit operator Rückgabetyp (Operand)
{
  // Anweisungen
}
```

für implizite Konvertierung und

```
public static explicit operator Rückgabetyp (Operand)
{
  // Anweisungen
}
```

für explizite Konvertierung verwendet. Operatoren können als Parameter nur Werteparameter übergeben werden, aber keine Parameter, welche die Schlüsselwörter `ref` oder `out` erwarten.

Der Unterschied zwischen einem impliziten und einem expliziten Operator besteht darin, dass ein impliziter Operator automatisch durch den Compiler angewendet werden kann, ein expliziter hingegen ausdrücklich im Quellcode angegeben werden muss.

So stellt beispielsweise die Umwandlung der Zahl 23 in einen String in der Zeile

```
string Text = "Hallo " + 23;
```

einen Aufruf des impliziten Konvertierungsoperators von `int` nach `string` dar. In der Regel reicht es aus, Operatoren als implizit du deklarieren. Es gibt jedoch Ausnahmefälle, in denen explizite Konvertierungsoperatoren verwendet werden müssen, beispielsweise wenn durch die Konvertierung ein Daten- oder Genauigkeitsverlust auftreten könnte.

Das folgende Beispiel implementiert die bereits bekannte Klasse zur Implementierung komplexer Zahlen und erweitert diese um einen Additions- und einen Konvertierungsoperator. Der Konvertierungsoperator berechnet den Abstand einer komplexen Zahl zum Nullpunkt und gibt diesen Wert als `double`-Wert zurück (s. Folgeseite).

```
public class CKomplexeZahl
{
  private double Real;
  private double Imaginaer;

  public CKomplexeZahl() : this(0, 0)
  {
  }

  public CKomplexeZahl(double Real, double Imaginaer)
  {
    this.Real = Real;
    this.Imaginaer = Imaginaer;
  }

  public void SetReal(double a)
  {
    Real = a;
  }

  public void SetImaginaer(double b)
  {
    Imaginaer = b;
  }

  public double GetReal()
  {
    return Real;
  }

  public double GetImaginaer()
  {
    return Imaginaer;
  }

  public static CKomplexeZahl operator + (CKomplexeZahl
ErsteZahl, CKomplexeZahl ZweiteZahl)
  {
    return new CKomplexeZahl(ErsteZahl.Real + ZweiteZahl.Real,
ErsteZahl.Imaginaer + ZweiteZahl.Imaginaer);
  }

  public static implicit operator double (CKomplexeZahl
EineZahl)
  {
    return (double)(Math.Sqrt(EineZahl.Real * EineZahl.Real +
EineZahl.Imaginaer * EineZahl.Imaginaer));
  }

  public void WriteLine()
  {
    Console.WriteLine("{0} + {1} * i", Real, Imaginaer);
  }
}
```

Code 44: Überladen von Operatoren

Natürlich könnte man diese Klasse noch um etliche weitere überladene Operatoren wie beispielsweise Multiplikation oder Vergleich erweitern.

Beim Überladen von Vergleichsoperatoren muss man allerdings beachten, dass sie immer paarweise überladen werden müssen. Das heißt, wird zum Beispiel der `==`-Operator überladen, so muss auch der `!=`-Operator überladen werden.

16.6 Ausblick

Im nächsten Kapitel werden Sie lernen, Fehler im Programmablauf zu erkennen und entsprechend zu reagieren.

17 Fehlerbehandlung

In diesem Kapitel werden Sie lernen, Fehler im Programmablauf zu erkennen und entsprechend zu reagieren.

17.1 Einleitung

Falls Sie bisher nicht nur die hier vorgestellten Programme nach-programmiert haben, sondern auch schon versucht haben, eigene Programme zu entwickeln, so sind Sie bestimmt schon einmal über den einen oder anderen Fehler gestolpert.

In jeder Programmiersprache gibt es drei Arten von Fehlern. Der erste Typ sind syntaktische Fehler, bei denen es sich meistens lediglich um einen Tippfehler handelt. Der Vorteil an syntaktischen Fehlern ist, dass sie vom Compiler entdeckt werden.

Der zweite Typ von Fehlern sind Laufzeitfehler, die erst bei der Ausführung des Programms auftreten und meistens bei der Kompilierung noch nicht erkannt werden können, da sie beispielsweise von Variablen, die der Benutzer verändern kann, abhängen. Beim Auftreten eines solchen Fehlers bricht die Ausführung des Programms üblicherweise ab, sofern der Fehler nicht innerhalb des Codes erkannt und abgefangen wird.

Der dritte Typ von Fehlern schließlich sind die sogenannten semantischen oder logischen Fehler, die sich gar nicht direkt bemerkbar machen, sondern im Lauf der Zeit Fehlfunktionen verursachen. Das hinterhältige an diesem Fehlertyp ist, dass seine Auswirkungen oft nicht dort auftreten, wo der Fehler stattgefunden hat, sondern häufig erst viel später und an einer ganz anderen Stelle im Code.

Da Fehler erster Art im Regelfall vom Compiler entdeckt werden und Fehler dritter Art nicht maschinell aufzuspüren sind, wird es in diesem Kapitel um die Erkennung und Behandlung von Laufzeitfehlern gehen.

17.2 Checked und unchecked

Einer der häufigsten Laufzeitfehler ist der Überlauf, bei dem versucht wird, eine Zahl in einer Variablen zu speichern, wobei aber die Größe der Zahl den Wertebereich der Variablen übersteigt. Standardmäßig ignoriert C# solche Fehler stillschweigend, um nicht bei jeder Kleinigkeit einen Programmabbruch zu verursachen.

Trotzdem kann es oft nützlich sein, auch Überlaufsfehler abzufangen. Dazu gibt es zwei Möglichkeiten. Entweder geben Sie beim Kompilieren den Parameter

```
/checked+
```

an und schalten so die Überlaufkontrolle für den gesamten Code ein, oder Sie können die zu überprüfenden Stellen direkt im Code in ein Paar geschweifter Klammern einschließen, das vom Schlüsselwort `checked` eingeleitet wird. Falls Sie mit Mono arbeiten, beachten Sie bitte, dass Sie den Parameter

```
-checked+
```

verwenden müssen.

Umgekehrt können Sie natürlich auch Programmabschnitte, die auf keinen Fall überprüft werden soll, mit dem Schlüsselwort `unchecked` kennzeichnen. Solche Blöcke werden dann - unabhängig davon, ob der Compilerschalter `/checked+` gesetzt ist oder nicht - nicht auf einen Überlauf hin kontrolliert.

Das folgende Programm berechnet die Fakultät einer Zahl, die als Kommandozeilenparameter übergeben wird. Die Fakultät ist dabei das Produkt einer Zahl mit allen kleineren Zahlen, die noch größer als Null sind. Beispielsweise ist die Fakultät von 5 das Produkt von 5 * 4 * 3 * 2 * 1 = 120. Bitte beachten Sie, dass die Klasse der Methode `Parse`, um einen String in einen `long`-Wert umzuwandeln, nicht *Long*, sondern *Int64* heißt.

```
using System;

public class Fakultaet
{
  public static void Main(string[] Argumente)
  {
    long Zahl = Int64.Parse(Argumente[0]);

    for(long i = (Zahl - 1); i > 0; i--)
      Zahl *= i;

    Console.WriteLine("Die Fakultät von {0} ist {1}.",
    Argumente[0], Zahl);
  }
}
```

Code 45: Fakultät berechnen

Wenn Sie diesem Programm als Parameter beispielsweise 5 übergeben, wird es als Ergebnis ganz korrekt 120 ausgeben. Übergeben Sie jedoch eine viel größere Zahl wie beispielsweise 10000, so wird bei einer der Multiplikationen das Ergebnis größer als die größte Zahl, die in einer `long`-Variablen gespeichert werden kann. In diesem Fall gibt das Programm als Ergebnis eine 0 aus.

```
using System;

public class Fakultaet
{
  public static void Main(string[] Argumente)
  {
    long Zahl = Int64.Parse(Argumente[0]);

    for(long i = (Zahl - 1); i > 0; i--)
      checked
      {
        Zahl *= i;
      }

    Console.WriteLine("Die Fakultät von {0} ist {1}.",
    Argumente[0], Zahl);
  }
}
```

Code 46: Fakultät berechnen mit 'checked'

Dieses Programm (Code 46, vorherige Seite) ist leicht verändert, denn die Multiplikation findet nun innerhalb eines `checked`-Blocks statt (die Klammern sind, da es sich nur um eine einzige Zeile handelt, eigentlich überflüssig) und der Überlauf wird nun bei der Ausführung des Programms erkannt und das Programm daraufhin mit der Ausnahme

```
Unhandled Exception: System.OverflowException: Exception
of type System.OverflowException was thrown.
```

abgebrochen. Selbstverständlich ist dieser Abbruch unter Verwendung einer Ausnahme genauso unelegant wie das ursprüngliche Verhalten (das Ignorieren des Überlaufs).

17.3 Ausnahmen abfangen

Glücklicherweise bietet C# jedoch eine Möglichkeit, Ausnahmefehler während der Laufzeit zu erkennen und abzufangen, so dass sie im Programm behandelt werden können. Dazu dienen die Anweisungen `try` und `catch`.

Das Schlüsselwort `try` leitet einen Block ein, der den kritischen Programmabschnitt enthält. Daran anschließend wird durch `catch` ein weiterer Block definiert, der nur im Falle einer Ausnahme ausgeführt wird. Die allgemeine Syntax sieht also folgendermaßen aus.

```
try
{
  // Anweisungen, die eine Ausnahme verursachen könnten
}
catch
{
  // Anweisungen zur Behandlung der Ausnahme
}
```

Dieser `catch`-Block würde jede Ausnahme abfangen, die innerhalb des `try`-Blocks auftreten könnte. Das ist aber oft nicht gewünscht, da nicht alle Ausnahmen vorhersehbar sind.

Beispielsweise könnte im catch-Block ein Überlauf einer Berechnung behandelt werden, tritt aber unerwarteterweise ein Datenträgerfehler auf, so kann das Programm nicht entsprechend reagieren.

Daher gibt man dem catch-Block üblicherweise noch die Art der Ausnahme als Parameter mit, auf die er reagieren soll. Alle anderen Ausnahmen, die dann nicht mehr abgefangen werden, werden einfach an die aufrufende Methode beziehungsweise im Falle der Main-Methode an das Betriebssystem durchgereicht. Nach Abarbeitung des catch-Blocks wird die Ausführung des Programms ganz normal fortgesetzt.

Als Parameter wird dem Schlüsselwort catch nicht nur der Typ der zu behandelten Ausnahme übergeben, sondern zusätzlich noch ein Variablenname, der die aufgetretene Ausnahme repräsentiert und durch den man beispielsweise nähere Informationen über den aufgetretenen Fehler abfragen kann. Der Name der Variablen kann dabei frei gewählt werden. Allerdings muss man beachten, dass die Variable nur gelesen, aber nicht verändert werden kann.

Die erweiterte Syntax für den catch-Block lautet also

```
catch(Ausnahmetyp Variablenname)
{
  // Anweisungen zur Behandlung der Ausnahme
}
```

Nun können Sie das vorangegangene Programm zur Berechnung der Fakultät so abändern, dass die Ausnahme erkannt und eine entsprechende Fehlermeldung ausgegeben wird. Sollte jedoch eine andere Ausnahme als ein Überlauf eintreten, so bricht das Programm die Ausführung dennoch wie gehabt ab, da für andere Ausnahmen keine Fehlerbehandlung definiert ist.

```
using System;

public class Fakultaet
{
  public static void Main(string[] Argumente)
  {
    long Zahl = Int64.Parse(Argumente[0]);

    for(long i = (Zahl - 1); i > 0; i--)
    {
      try
      {
        checked
          Zahl *= i;
      }
      catch(OverflowException e)
      {
        Console.WriteLine("Es ist ein Überlauf aufgetreten!");
        return;
      }
    }

    Console.WriteLine("Die Fakultät von {0} ist {1}.",
    Argumente[0], Zahl);
  }
}
```

Code 47: Fakultät berechnen mit 'try-/catch-Block'

Um eine allgemeine Ausnahme abzufangen, deren Typ Ihnen nicht bekannt ist, geben Sie einfach *Exception* ohne nähere Spezifikation als Typ an. Sie können beide Techniken auch kombinieren und zunächst spezielle Behandlungsroutinen für bestimmte Ausnahmen schreiben und alle weiteren von einer allgemeinen Routine abfangen lassen, indem Sie mehrere `catch`-Blöcke hintereinander setzen.

```
catch(OverflowException e)
{
  // Anweisungen zur Behandlung eines Überlaufs
}
catch
{
  // Anweisungen zur Behandlung aller sonstigen Ausnahmen
}
```

Falls Sie mehrere `catch`-Blöcke einsetzen, müssen Sie darauf achten, dass Sie die Routinen zur Behandlung der allgemeineren Ausnahmen weiter unten platzieren. Wenn Sie beispielsweise

```
catch
{
  // Anweisungen zur Behandlung aller möglichen Ausnahmen
}
catch(OverflowException e)
{
  // Anweisungen zur Behandlung eines Überlaufs
}
```

als Code verwenden, wird der zweite `catch`-Block nie erreicht, da der erste alle Ausnahmen behandelt und somit auch den Überlauf bereits beinhaltet.

Die folgende Tabelle (s. Folgeseite) enthält einen Überblick über häufig auftretende Typen von Ausnahmen. Falls Sie nähere Informationen zu den verschiedenen Typen von Ausnahmen benötigen, finden Sie diese in der Dokumentation zu .net.

Klasse	*Beschreibung*
Exception	Basisklasse für alle Ausnahmen
SystemException	Basisklasse für alle zur Laufzeit generierten Ausnahmen
IndexOutOfRangeException	Signalisiert eine Bereichsverletzung für einen Arrayindex
NullReferenceException	Signalisiert den Zugriff auf eine null-Referenz
InvalidOperationException	Wird von verschiedenen Methoden signalisiert, wenn die Operation für den aktuellen Objektzustand nicht durchgeführt werden kann
ArgumentException	Basisklasse für alle Ausnahmen, die auf Argumentfehler zurückgehen
ArgumentNullException	Wird von Methoden signalisiert, wenn für einen Parameter unerwartet das Argument null vorliegt
ArgumentOutOfRangeException	Wird von Methoden signalisiert, wenn für das Argument eines Parameters eine Bereichsverletzung vorliegt

Tabelle 13: Typen von Ausnahmen

17.4 Aufräumcode

Unabhängig davon, ob eine Ausnahme aufgetreten ist oder nicht, kann es manchmal notwendig sein, abschließende Operationen durchzuführen. Beispielsweise sollte eine Datei von Ihrer Anwendung wieder geschlossen werden, egal, ob aus ihr gelesen werden konnte oder nicht.

Dazu dient das Schlüsselwort `finally`, das nach dem letzten `catch` einen weiteren Block einleitet, der immer ausgeführt wird. Der Sinn von `finally` offenbart sich vor allem dadurch, dass es auch gänzlich ohne `catch`-Blöcke eingesetzt werden kann, nur in Verbindung mit einem `try`-Block.

So können also beispielsweise sogar fehlgeschlagene Operationen sauber beendet werden, die Sie nicht selbst abfangen, sondern bei denen das Programm durch das Betriebssystem abgebrochen wird. Die Syntax entspricht daher fast der bereits bekannten.

```
try
{
  // Anweisungen, die eine Ausnahme verursachen könnten
}
catch
{
  // Anweisungen zur Behandlung der Ausnahme
}
finally
{
  // Anweisungen zum Aufräumen
}
```

Das Programm zur Berechnung der Fakultät kann damit beispielsweise so umgeschrieben werden, dass je nachdem, ob eine Ausnahme aufgetreten ist oder nicht, im `finally`-Block entweder das Ergebnis oder eine Fehlermeldung ausgegeben wird, wie folgendes Beispiel (s. Folgeseite) zeigt.

```
using System;

public class Fakultaet
{
  public static void Main(string[] Argumente)
  {
    bool AusnahmeAufgetreten = false;
    long Zahl = Int64.Parse(Argumente[0]);

    for(long i = (Zahl - 1); i > 0; i--)
    {
      try
      {
        checked
          Zahl *= i;
      }
      catch(OverflowException e)
      {
        AusnahmeAufgetreten = true;
      }
      finally
      {
        if(!AusnahmeAufgetreten)
          Console.WriteLine("Die Fakultät von {0} ist {1}.",
           Argumente[0], Zahl);
        else
          Console.WriteLine("Es ist ein Überlauf
           aufgetreten!");
      }
    }
  }
}
```

Code 48: Fakultät berechnen mit 'try-/catch- und finally-Block'

17.5 Ausnahmen auslösen

Die Ausführung eines Programms ist übrigens nicht darauf beschränkt, dass Ausnahmen nur vom System bei Laufzeitfehlern ausgelöst werden. Sie können auch programmieren, dass eine Ausnahme ausgelöst wird, um beispielsweise den Benutzer auf einen Fehler hinzuweisen.

```
throw new ArgumentException("Es müssen mindestens zwei
Parameter angegeben werden!");
```

Wie Sie sehen, dient dazu das Schlüsselwort `throw`, dem Sie lediglich eine neue Instanz einer beliebigen Ausnahmeklasse übergeben müssen. Der Parameter definiert dabei den Text, der dem Benutzer als Ausnahmegrund angezeigt wird.

Außerdem können Sie mittels des Schlüsselworts `throw` auch eine aufgetretene Ausnahme, die Sie in einem `catch`-Block behandeln, an die übergeordnete Methode beziehungsweise das Betriebssystem weiterleiten, indem Sie throw innerhalb eines `catch`-Blocks ohne Parameter aufrufen.

17.6 Eigene Ausnahmen definieren

Selbstverständlich können Sie auch eigene Ausnahmen definieren, indem Sie einfach eine bestehende Klasse für Ausnahmen ableiten und den Konstruktor, der als einzigen Parameter den auszugebenden Text als string erwartet, überschreiben. Allerdings sollten Sie damit sparsam umgehen, da zu viele benutzerdefinierte Ausnahmen eher verwirren.

Falls Sie aber dennoch eine eigene Ausnahmeklasse entwickeln möchten, sollten Sie sich zumindest bei der Namensgebung der Klasse daran halten, dass der Name auf *Exception* endet, so wie es auch bei allen vorgefertigten Ausnahmeklassen von .net der Fall ist.

17.7 Ausblick

Im nächsten Kapitel werden Sie lernen, was Namensräume sind und wie sie in C# benutzt werden.

18 Namensräume

In diesem Kapitel werden Sie lernen, was Namensräume sind und wie sie in C# benutzt werden.

18.1 Einleitung

In den bisherigen Kapiteln haben Sie fast alle Grundlagen der Programmiersprache C# erlernt. Dieses Kapitel soll die Einführung in die Grundlagen abschließen.

Alle Programme, die Sie je in C# geschrieben haben, hatten alle eine Zeile Code gemeinsam, nämlich die erste.

```
using System;
```

Im zweiten Kapitel haben Sie gelernt, dass Sie diese Zeile verwenden können, um sich die Schreibarbeit zu vereinfachen - statt `System.Console.WriteLine` konnten Sie `Console.WriteLine` verwenden. Doch diese Zeile bedeutet natürlich etwas mehr, als nur eine Vereinfachung der Schreibweise.

System ist ein sogenannter `Namensraum` (auch *Namespace* genannt). Namensräume dienen dazu, Klassen mit ähnlicher Bedeutung in einen gemeinsamen Behälter - eben den Namensraum - zusammenzufassen. So enthält beispielsweise der Namensraum `System` nicht nur die Klasse `Console` zum Zugriff auf Ein- und Ausgabegeräte, sondern auch Klassen zur Fehlerbehandlung (die Klasse `Exception` und ihre Varianten, die Sie im letzten Kapitel kennen gelernt haben) oder auch die Klasse `Math`.

Es gibt innerhalb von .net noch sehr viele weitere Namensräume, die Sie im Weiteren teilweise noch kennen lernen werden.

Wie bereits angesprochen, können Sie sich den Zugriff auf die Klassen innerhalb eines Namensraumes vereinfachen, indem Sie ihn mittels des Schlüsselwortes `using` innerhalb Ihres Programms bekannt geben. So müssen Sie nicht immer den vollqualifizierten Namen der Klasse verwenden.

Namensräume sind in C# hierarchisch aufgebaut, so ist beispielsweise der Namensraum `System.Collections` unterhalb von `System` angeordnet. Wie Sie sehen, dient auch hier der Punkt-Operator wieder als Trennzeichen. Durch die Möglichkeit zu Hierarchien lassen sich Namensräume sehr übersichtlich strukturieren.

18.2 Eigene Namensräume

Außer der Verwendung von vorgefertigten Namensräumen können Sie auch Ihre eigenen erstellen, indem Sie innerhalb einer Datei alle thematisch zusammengehörenden Klassen in einen Block einfassen, der von dem Schlüsselwort `namespace` (gefolgt von dem Namen des Namensraums) eingeleitet wird.

Die Syntax lautet also

```
namespace Namensraum
{
  // Alle Klassen, die im Namensraum enthalten sein
     sollen
}
```

Diese Namensräume können Sie dann wie die vorgefertigten verwenden. Einzige Bedingung dafür ist, dass sich die Dateien, die die entsprechenden Namensräume enthalten, in dem selben Ordner befinden wie das Programm, das auf sie zugreifen möchte. Innerhalb der Programmbeispiele von diesem Guide wurden keine selbstdefinierten Namensräumen verwendet.

Dennoch sollten Sie sich angewöhnen, mit Namensräumen zu arbeiten, da es vor allem bei etwas größeren Programmen die Übersichtlichkeit und Wart-barkeit deutlich erhöht.

18.3 Ausblick

Herzlichen Glückwunsch! Sie haben guide to C# nun vollständig durchgearbeitet und kennen alle Grundlagen der Programmiersprache C#, so dass Sie sich auch mit weitergehenden Themen wie beispielsweise der Programmierung von grafischen Oberflächen oder Datenbanken beschäftigen können.

Wir wünschen Ihnen dabei viel Erfolg!

Windows Forms

19 Windows Forms - Erste Schritte

In diesem Kapitel werden Sie die Grundlagen zur Entwicklung grafischer Oberflächen mittels .net kennen lernen.

19.1 Einleitung

Allen Programme, die Sie bisher entwickelt haben, war gemeinsam, dass sie über keine grafische Oberfläche verfügten, sondern nur im Textmodus lauffähig waren. Solche Programme bezeichet man unter .net als *Konsolenapplikationen*.

Leider sind Konsolenapplikationen in ihrem Funktionsumfang sehr beschränkt, so können beispielsweise weder eigene Fenster geöffnet, noch Bilder oder Grafiken dargestellt, noch die Maus genutzt werden. Als Lösung bieten sich grafische Oberflächen, so genannte *GUIs* (Graphical user interface), an.

19.2 Windows Forms

Um die Entwicklung von GUIs in .net möglichst einfach zu gestalten, enthält die .net Klassenbibliothek eine sehr leistungsfähige Komponente namens *Windows Forms*. Diese Komponente bietet zahlreiche Klassen und Methoden, um Fenster und Steuerelemente zu verwalten, auf die Maus zuzugreifen, mit verschiedenen Schriftarten und Farben zu arbeiten, Grafiken zu zeichnen, ...

Fast die gesamte Funktionalität von Windows Forms ist in drei DLLs enthalten, die beim Kompilieren als zusätzliche Referenzen eingebunden werden müssen.

DLLs für Windows Forms
System.dll
System.Drawing.dll
System.Windows.Forms.dll

Tabelle 14: DLLs für Windows Forms

Um diese DLLs einzubinden, müssen Sie beim Kompilieren den Parameter `/reference` (oder kurz `/r`) verwenden, gefolgt von einem Doppelpunkt und den durch Kommata getrennten Dateinamen der DLLs, die eingebunden werden sollen. Falls Sie mit Mono arbeiten, verwenden Sie bitte den Parameter `-reference` beziehungsweise `-r`.

19.3 Hallo Welt GUI

Da Sie nun über die notwendigen Grundlagen verfügen, können Sie Ihre erste grafische .net-Applikation entwickeln, die in Anlehnung an das klassische *Hallo Welt*-Programm eine Meldung mit dem *Text Hallo Welt!* anzeigen soll.

Um eine solche Meldung darzustellen, können Sie die Klasse `System.Windows.Forms.MessageBox` verwenden. Da Meldungsfenster sehr häufig benötigt werden, enthält diese Klasse eine statische Methode namens `Show`, mit der Sie Meldungsfenster direkt anzeigen können, ohne vorher eine Instanz der Klasse erstellen zu müssen. Der anzuzeigende Text wird der Methode dabei einfach als Parameter übergeben.

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

public class HalloWelt
{
  public static void Main()
  {
    MessageBox.Show("Hallo Welt!");
  }
}
```

Code 49: Hallo Welt! mit Windows Forms

Wenn Sie dieses Programm kompilieren und ausführen, wird auf dem Bildschirm ein Meldungsfenster angezeigt, welches den *Text Hallo Welt!* sowie einen *OK*-Button enthält, mit dem das Fenster wieder geschlossen werden kann.

Beachten Sie bitte außerdem, dass diese Methode die Programmausführung so lange anhält, bis das Meldungsfenster wieder vom Benutzer geschlossen wird. Solche Fenster, die den weiteren Ablauf des Programms blockieren, bis sie wieder geschlossen werden, nennt man *modale Fenster*.

Das Gegenteil von modalen Fenstern sind *nicht-modale Fenster*, die parallel zum eigentlichen Programm geöffnet werden und deren `Show`-Methode sofort zur Stelle des Aufrufs zurückkehrt.

Ein bekanntes Beispiel für ein modales Fenster ist der *Datei öffnen*-Dialog der meisten Anwendungen. So lange dieses Dialogfenster geöffnet ist, kann nicht in der jeweiligen Anwendung weitergearbeitet werden. Ein nicht-modales Fenster dagegen ist beispielsweise der *Suchen und Ersetzen*-Dialog von Microsoft Word, da unabhängig davon, ob das Fenster noch geöffnet ist oder nicht, weiterhin Text editiert werden kann.

Bisher haben Sie der `Show`-Methode nur den anzuzeigenden Text als Parameter übergeben. Allerdings gibt es noch weitere Überladungen dieser Methode, die mehr Parameter erwarten und dementsprechend auch mehr Möglichkeiten bieten.

Als zweiten Parameter können Sie nämlich einen weiteren String übergeben, der den Text enthält, welcher in der Titelleiste des Meldungsfensters angezeigt werden soll. Hier wird oft der Name der Anwendung oder die Funktion eingesetzt, in deren Rahmen die Meldung auftritt, damit der Benutzer das Meldungsfenster eindeutig einem Programm zuordnen kann.

So könnte beispielsweise die bisherige Zeile

```
MessageBox.Show("Hallo Welt!");
```

durch

```
MessageBox.Show("Hallo Welt!", "Hinweis");
```

ersetzt werden, um den Text Hinweis als Titel zu verwenden.

Als dritten Parameter können Sie einen Wert angeben, der bestimmt, welche Buttons statt des standardmäßigen OK-Buttons angezeigt werden sollen. Die möglichen Werte sind in der Aufzählung
`System.Windows.Forms.MessageBoxButtons`
enthalten.

Wert	***Angezeigte Buttons***
MessageBoxButtons.OK	OK
MessageBoxButtons.OKCancel	OK, Abbrechen
MessageBoxButtons.YesNo	Ja, Nein
MessageBoxButtons.YesNoCancel	Ja, Nein, Abbrechen
MessageBoxButtons.RetryCancel	Wiederholen, Abbrechen
MessageBoxButtons.AbortRetryIgnore	Abbrechen, Wiederholen, Ignorieren

Tabelle 15: Werte für Buttons

Falls Sie mehrere Buttons in einem Meldungsfenster verwenden, ist es oft notwendig, herauszufinden, welcher Button vom Benutzer angeklickt wurde. Dazu liefert die Methode `Show` einen entsprechenden Rückgabewert vom Typ `System.Windows.Forms.DialogResult`.

Wert	**Angeklickter Button**
DialogResult.None	Das Meldungsfenster wurde nicht durch einen Klick auf einen Button geschlossen.
DialogResult.OK	OK
DialogResult.Cancel	Abbrechen
DialogResult.Yes	Ja
DialogResult.No	Nein
DialogResult.Abort	Abbrechen
DialogResult.Retry	Wiederholen
DialogResult.Ignore	Ignorieren

Tabelle 16: Dialog Results

Mit dem vierten Parameter können Sie festlegen, dass innerhalb des Meldungsfensters neben dem Text noch ein Icon angezeigt werden soll. Dabei können Sie allerdings keine eigenen Icons gestalten, sondern nur einige wenige, im Betriebssystem definierte, verwenden.

Der Wert, durch den das anzuzeigende Icon bestimmt wird, ist dabei vom Typ `System.Windows.Forms.MessageBoxIcon`. Die gültigen Werte sind in der folgenden Tabelle aufgelistet.

Wert	**Angezeigtes Icon**
MessageBoxIcon.None	Es wird kein Icon angezeigt.
MessageBoxIcon.Question	Fragezeichen
MessageBoxIcon.Information	Kleines *i*
MessageBoxIcon.Warning	Ausrufezeichen
MessageBoxIcon.Error	Weißes *X* im roten Kreis

Tabelle 17: Buttonwerte und dazugehörige Icons

Der folgende Codeausschnitt demonstriert, wie mit Hilfe einer MessageBox eine Abfrage realisiert werden könnte, ob eine geänderte Datei gespeichert werden soll oder nicht.

```
DialogResult dr;
dr = MessageBox.Show("Möchten Sie die Datei speichern?",
"Datei speichern", MessageBoxButtons.YesNoCancel,
MessageBoxIcon.Question);
select(dr)
{
  case DialogResult.Yes:
    // Datei speichern
    break;
  case DialogResult.No:
    // Datei nicht speichern
    break;
  case DialogResult.Cancel:
    // Abbrechen
    break;
}
```

Code 50: Codeausschnitt zum Abfragen einer Dateispeicherung

Natürlich hätten Sie an Stelle des Objektes dr zur temporären Speicherung des Rückgabewertes den Aufruf von MessageBox.Show auch direkt in die select-Anweisung schreiben können.

Schließlich gibt es noch einen fünften Parameter der Methode Show, mit dessen Hilfe man festlegen kann, welcher der angezeigten Buttons der Standardbutton sein soll (der Standardbutton wird ausgelöst, wenn der Benutzer die Taste Enter drückt).

Dieser Parameter ist vom Typ
System.Windows.Forms.MessageBoxDefaultButton
und kann einen der folgenden Werte annehmen.

Werte für MessageBoxDefaultButton
MessageBoxDefaultButton.Button1
MessageBoxDefaultButton.Button2
MessageBoxDefaultButton.Button3

Tabelle 18: Werte für MessageBoxDefaultButton

Wird dieser Parameter nicht angegeben, so wird automatisch der erste Button als Standardbutton verwendet. Da diese Belegung in den meisten Fällen sinnvoll ist, wird der Parameter `MessageBoxDefaultButton` in der Praxis allerdings nur selten eingesetzt.

Insgesamt sind Meldungsfenster also eine sehr einfache und komfortable Möglichkeit, um kurze Nachrichten zur Information oder zur Bestätigung anzuzeigen.

19.4 Eigene Fenster

Natürlich reicht ein simples Meldungsfenster nicht aus, um eine Anwendung als eine Anwendung, die mit einer grafischen Oberfläche ausgestattet ist, bezeichnen zu können.

Um eine vollwertige grafische Oberfläche zu gestalten, benötigt man Fenster, die sich beispielsweise durch eine Menü-, eine Symbol- und eine Statusleiste ergänzen lassen und in denen man nach Belieben komplexe grafische und textuelle Ausgaben erzeugen kann.

Für diese Art von Fenstern gibt es innerhalb von Windows Forms die Klasse `System.Windows.Forms.Form`. Im Gegensatz zu `MessageBox` sind die Methoden dieser Klasse jedoch nicht statisch, sondern es müssen Instanzen erzeugt werden. Jede Instanz stellt dann ein eigenes Fenster dar, das frei gestaltet werden kann.

Der Konstruktor der Klasse `Form` erwartet keinen Parameter, so dass ein Aufruf von

```
Form MyForm = new Form();
```

genügt, um ein neues Fenster anzulegen.

Allerdings wird das Fenster nicht automatisch angezeigt (man könnte es ja auch nur zur späteren Verwendung erstellen wollen), so dass hierzu noch ein Aufruf der Instanzenmethode `Show` notwendig ist.

Alternativ zu einem Aufruf von `Show` können Sie auch schreibend auf die Eigenschaft `Visible` eines Fensters zugreifen und ihr den Wert `true` zuweisen. In der Tat macht die Methode `Show` intern nichts anderes, als diese Eigenschaft zu ändern.

Um ein Fenster wieder zu schließen, können Sie dementsprechend entweder die Methode `Hide` aufrufen oder die Eigenschaft `Visible` auf den Wert `false` zurücksetzen.

19.5 Titelleiste

Um nun für ein dermaßen erzeugtes Fenster eine Titelleiste zu erstellen, können Sie die Eigenschaft `Text` verwenden, der Sie einen String zuweisen und die Sie auch wieder auslesen können.

Das folgende Programm legt also ein Objekt der Klasse `Form` an, weist ihm einen Text für seine Titelleiste zu, und zeigt es schließlich als Fenster an.

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

public class HalloWelt
{
  public static void Main()
  {
    Form MyForm = new Form();
    MyForm.Text = "Hallo Welt";
    MyForm.Show();
  }
}
```

Code 51: Fenster mit Titelleiste

Wenn Sie das Programm kompilieren und ausführen, werden Sie feststellen, dass das Fenster - wenn überhaupt - nur für den Bruchteil einer Sekunde sichtbar ist und sofort wieder geschlossen wird.

Das liegt daran, dass die Methode `Show` sofort nach dem Anzeigen des Fensters wieder zum Aufrufer - in diesem Fall also zur `Main`-Methode - zurückspringt. Da diese aber direkt nach dem Aufruf von `Show` endet, wird das Programm und damit alle seine Fenster geschlossen.

19.6 Application.Run

Die Lösung liegt in der statischen Methode `Run` der Klasse `System.Windows.Forms.Application`. Da diese Methode ein elementares Konzept von grafischen Anwendungen darstellt, soll an dieser Stelle etwas näher auf sie eingegangen werden.

Neben den Unterschieden zwischen grafischen und Konsolenanwendungen, die Sie bisher bereits kennengelernt haben, gibt es noch einen weiteren, weniger auffäliigen, aber sehr elementaren Unterschied. Während Konsolenanwendungen immer entweder mit der Ausführung des Programms beschäftigt sind oder auf eine Eingabe vom Benutzer warten, findet beides bei grafischen Applikationen (mehr oder weniger) gleichzeitig statt.

Dieses Verhalten können Sie bei jeder grafischen Applikation beobachten, denn es ist fast immer möglich, beispielsweise ein Menü zu aktivieren, unabhängig davon, ob die Software gerade mit einer Berechnung beschäftigt ist oder nicht.

Um diese Fähigkeit zu besitzen, enthält eine grafische Applikation eine Schleife (namens *Application message loop*), die während der gesamten Laufzeit des Programms im Hintergrund aktiv ist und ständig überprüft, ob der Benutzer eine Aktion wie beispielsweise einen Tastendruck oder einen Mausklick ausgeführt hat.

Tritt ein solcher Fall ein, ruft die Schleife eine entsprechende Methode zur Behandlung dieses Ereignisses auf, ansonsten wartet sie weiter und das Programm kann ungestört mit seinen internen Aktiviväten fortfahren.

Zum Glück müssen Sie die Application message loop aber nicht selbst programmieren, denn Sie wird Ihnen vom Betriebssystem vorgefertigt zur Verfügung gestellt. Zur Steuerung dieser Schleife dient die bereits erwähnte Klasse namens `Application`. Indem Sie die Methode `Run` aufrufen, initialisieren Sie eine neue Application message loop und fügen diese Ihrer Anwendung hinzu.

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

public class HalloWelt
{
  public static void Main()
  {
    Form MyForm = new Form();
    MyForm.Text = "Hallo Welt";
    MyForm.Show();
    Application.Run();
  }
}
```

Code 52: Fenster mit Titelleiste und der Methode Run

Nachdem Sie das vorige Programm nun um die Anweisung `Application.Run();` ergänzt haben, arbeitet die Anwendung nun anscheinend wie gewünscht. Sobald man das Programm startet, wird das Fenster angezeigt, es wird aber nicht mehr sofort wieder geschlossen.

Allerdings arbeitet die Anwendung nur scheinbar wie gewünscht, denn auch wenn Sie das Fenster irgendwann schließen, wird das Programm nicht beendet. Dies liegt daran, dass die Application message loop noch immer aktiv ist und nicht weiß, dass das Schließen des Fensters dem Beenden der Anwendung entspricht. Eine solchermaßen hängengebliebene Anwendung können Sie nur noch mittels *Ctrl* + *C* beenden.

Um mit dem Schließen des Fensters zeitgleich auch das Programm zu beenden, müssten Sie die statische Methode `Exit` der Klasse `Application` aufrufen, die das Gegenstück zur `Run`-Methode darstellt und eine Application message loop wieder beendet.

Das Problem bei dieser Vorgehensweise ist allerdings, dass es einen relativ hohen Aufwand darstellen würde, das Ereignis *Fenster schließen* abzufangen und mittels eines Aufrufs von `Application.Exit();` die Application message loop zu beenden.

Deshalb gibt es die Möglichkeit, der `Run`-Methode das Hauptfenster als Parameter zu übergeben. Wird dieses dann irgendwann geschlossen, so wird auch die Application message loop beendet und somit in die `Main`-Methode zurückgekehrt, die das Programm dann beendet.

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

public class HalloWelt
{
  public static void Main()
  {
    Form MyForm = new Form();
    MyForm.Text = "Hallo Welt";
    Application.Run(MyForm);
  }
}
```

Code 53: Fenster mit Titelleiste und Parameterübergabe

Wenn Sie dieses Programm nun kompilieren und ausführen, werden Sie feststellen, dass es endlich wie gewünscht arbeitet und beim Schließen des Fensters auch die Anwendung beendet wird. Beachten Sie bitte, dass kein Aufruf von `MyForm.Show();` mehr nötig ist, denn das Anzeigen des Fensters wird von der `Run`-Methode automatisch durchgeführt.

Nachdem dieses Programm nun wie gewünscht arbeitet, bleibt noch ein kleiner Schönheitsfehler, den es zu beheben gilt. Falls Sie das Programm nämlich nicht von der Kommandozeile starten, sondern über eine grafische Oberfläche, wird dennoch ein Konsolenfenster geöffnet, auch wenn hierin nichts aufgegeben wird.

Das liegt daran, dass es sich bei diesem Programm noch immer um eine Konsolenanwendung (die einfach nur ein zusätzliches Fenster öffnet) handelt und nicht um eine grafische Anwendung. Um .net mitzuteilen, dass es sich um eine rein grafische Anwendung handelt und kein Konsolenfenster benötigt wird, können Sie dem Compiler den Parameter

```
/target:winexe
```

(oder kurz `/t:winexe`) übergeben. Falls Sie mit Mono arbeiten, verwenden Sie bitte den Parameter `-target:winexe`.

19.7 Ausblick

Im nächsten Kapitel werden Sie lernen, welche allgemeinen Eigenschaften und Methoden es in der Klasse `Form` gibt und wie Sie diese einsetzen können.

20 Fenster

In diesem Kapitel werden Sie lernen, welche allgemeinen Eigenschaften und Methoden es in der Klasse `Form` gibt und wie Sie diese einsetzen können.

20.1 Einleitung

Im letzten Kapitel haben Sie gelernt, dass Sie eigene Fenster erstellen können, indem Sie ein Objekt der Klasse `Form` anlegen. In der Praxis ist es aber oft nützlich, nicht nur ein Objekt, sondern eine vollständige Klasse zur Verfügung zu haben.

Natürlich müssen Sie diese Klasse nicht komplett selbst erstellen, sondern Sie können einfach eine neue Klasse von `Form` ableiten. Die neue Klasse erbt dann alle Eigenschaften und Methoden der übergeordneten Klasse, Sie können sie allerdings nach Belieben erweitern.

20.2 Form vererben

Wenn Sie eine eigene Klasse von der `Form`-Klasse ableiten, beachten Sie bitte, dass in der `Main`-Methode ein Objekt dieser neuen Klasse angelegt und an `Application.Run()` als Parameter übergeben wird.

Außerdem können alle nötigen Intialisierungen - wie beispielsweise das Zuweisen eines Textes an die Eigenschaft `Text` - nun von der Methode `Main` in den Konstruktor der Klasse verlagert werden.

Das folgende Programm (s. Folgeseite) zeigt ein leeres Fenster an und weist seiner Titelleiste den Text *Hallo Welt* zu.

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

public class HalloWelt : Form
{
  public HalloWelt()
  {
    this.Text = "Hallo Welt";
  }

  public static void Main()
  {
    Application.Run(new HalloWelt());
  }
}
```

Code 54: Anlegen einer Klasse

Dieses Verfahren - eine Klasse an Stelle eines Objektes anzulegen - hat mehrere Vorteile. Zum einen können Sie nötige Initialisierungen an das Fenster binden, indem Sie sie in den Konstruktor verlagern. Jedes Mal, wenn Sie nun ein Objekt dieser Klasse anlegen, erhalten Sie ein Fenster, welches bereits entsprechend initialisiert ist.

Zum anderen können Sie Methoden der Superklasse überschreiben und so das Verhalten des Fensters verändern oder erweitern. Diese Technik werden Sie im nächsten Kapitel genauer kennen lernen, sobald Sie beginnen, Ausgaben in einem Fenster vorzunehmen.

20.3 Die Main-Methode

Wenn Sie sich den Code noch einmal ansehen, wundern Sie sich eventuell über die scheinbar merkwürdige Position der `Main`-Methode. Immerhin befindet sie sich in eben der Klasse, von der sie ein Objekt anlegen soll. Wie also kann die `Main`-Methode ausgeführt werden, wenn die Klasse, welche sie enthält, noch gar nicht existiert?

Die Lösung liegt in dem Schlüsselwort `static`. Dadurch gehört die Methode zwar zu der Klasse `HalloWelt`, sie ist aber nicht an eine Instanz eines Objektes gebunden, sondern existiert - unabhängig von der Anzahl der von dieser Klasse erstellten Objekte - immer genau ein Mal.
Rein theoretisch können Sie die `Main`-Methode auch in eine eigene Klasse auslagern, allerdings empfiehlt sich dies nicht. In der Regel wird die Klasse, welche das Hauptfenster definiert, auch die `Main`-Methode enthalten.

20.4 Fenstergröße

Als nächstes werden Sie lernen, wie Sie die Größe eines Fensters verändern können. Dazu existieren drei Eigenschaften, die Sie sowohl auslesen als auch schreiben können.

Die ersten beiden Eigenschaften sind `Width` und `Height`, welche die Breite und die Höhe eines Fensters in Pixeln beschreiben. Sie können die beiden Eigenschaften vollständig unabhängig voneinander verwenden, das heißt, Sie können beispielsweise die Breite verändern, ohne auf die Höhe zugreifen zu müssen.

Die Zahlen, die Sie `Width` und `Height` zuweisen (oder auslesen), sind vom Datentyp `int` und beschreiben die Gesamtbreite und -höhe des Fensters, also unter anderem inklusive Rahmen und Titelleiste.

Um einem Fenster nun beispielsweise eine Breite von 300 und eine Höhe von 150 Pixeln zuzuweisen, können Sie folgende Anweisungen verwenden.

```
MyForm.Width = 300;
MyForm.Height = 150;
```

Selbstverständlich können Sie auch relative Angaben verwenden. Um also beispielsweise die derzeitige Breite eines Fensters zu verdoppeln, benutzen Sie folgende Anweisungen.

```
int Breite = MyForm.Width;
Breite *= 2;
MyForm.Width = Breite;
```

Bitte beachten Sie, dass Sie innerhalb einer Anweisung niemals auf eine Eigenschaft gleichzeitig lesend und schreibend zugreifen dürfen! Die folgende Zeile wäre demnach unzulässig - auch wenn sie syntaktisch korrekt aussieht.

```
MyForm.Width = MyForm.Width * 2;
```

Die dritte Eigenschaft, mittels derer Sie auf die Größe eines Fensters zugreifen können, heißt `Size` und enthält sowohl die Breite als auch die Höhe. Dazu reicht der Datentyp `int` natürlich nicht aus, so dass Size vom Typ `System.Drawing.Size` ist.

Um auf die Breite und die Höhe getrennt zugreifen zu können, enthält die Eigenschaft `Size` wiederum zwei Eigenschaften, nämlich `Width` und `Height`.

Um beispielsweise einem Fenster mittels `Size` eine Größe von 300 mal 150 Pixeln zuzuweisen, können Sie die folgenden Anweisungen verwenden.

```
Size size = new Size();
size.Width = 300;
size.Height = 150;
MyForm.Size = size;
```

Alternativ können Sie beim Anlegen eines Objektes von der Klasse `Size` auch dem Konstruktor die Breite und Höhe als Parameter übergeben.

```
MyForm.Size = new Size(300, 150);
```

Außer der Klasse `System.Drawing.Size` existiert noch eine weitere Klasse für Größen, nämlich `System.Drawing.SizeF`, die auch Fließkommazahlen verarbeiten kann. Diese Klasse werden Sie in späteren Kapiteln noch benötigen.

20.5 Fensterposition

Ähnlich der Größe können Sie auch die Position eines Fensters mittels Eigenschaften ermitteln und verändern. Für die Position gibt es aber sogar sechs Eigenschaften, die Sie verwenden können.

Die ersten beiden, nämlich `Top` und `Left` kennzeichnen den oberen und den linken Rand eines Fensters bezüglich des Desktops. Die linke, obere Ecke des Desktops wird dabei mit den Koordinaten (*0*, *0*) gekennzeichnet, nach rechts und unten werden die Koordinaten jeweils größer.

Um ein Fenster (genauer gesagt, seine linke, obere Ecke) beispielsweise jeweils 100 Pixel vom linken und oberen Rand des Desktops entfernt zu positionieren, können Sie die folgenden Zeilen verwenden.

```
MyForm.Top = 100;
MyForm.Left = 100;
```

Mittels der beiden nächsten Eigenschaften können Sie feststellen, wie weit der rechte und untere Rand eines Fensters von der rechten, unteren Ecke des Desktops entfernt ist. Diese beiden Eigenschaften heißen `Right` und `Bottom`.

Beachten Sie bitte, dass diese beiden Eigenschaften nicht verändert, sondern nur gelesen werden können. Um ihre Werte zu beeinflussen, müssen Sie die `Left`, `Top`, `Width` oder `Height` verändern.

Wie bei der Größe gibt es auch für die Position eine Eigenschaft, die sowohl den linken als auch den oberen Rand gemeinsam enthält. Diese Eigenschaft heißt `DesktopLocation` und ist vom Datentyp `System.Drawing.Point`, welcher wiederum zwei Eigenschaften namens `X` und `Y` enthält.

Um also ein Fenster 100 Pixel vom oberen und 200 Pixel vom linken Rand des Desktops entfernt anzuzeigen, können Sie die folgenden Zeilen verwenden.

```
Point point = new Point();
point.X = 200;
point.Y = 100;
MyForm.DesktopLocation = point;
```

Alternativ können Sie beim Anlegen eines Objektes von der Klasse `Point` auch dem Konstruktor die X- und Y-Koordinate als Parameter übergeben.

```
MyForm.DesktopLocation = new Point(200, 100);
```

Außer der Klasse `System.Drawing.Point` existiert noch eine weitere Klasse für eine Position, nämlich `System.Drawing.PointF`, die auch Fließkommazahlen verarbeiten kann. Auch diese Klasse werden Sie in späteren Kapiteln noch benötigen.

Die letzte Eigenschaft, mit der Sie die Position eines Fensters bestimmen können, heißt `StartPosition`. Mit ihr können Sie festlegen, an welcher Stelle des Bildschirms das Fenster beim Öffnen erscheint.

Als Parameter erwartet `StartPosition` einen Wert der Enumeration `System.Windows.Forms.FormStartPosition`

Wert	***Größe / Position***
FormStartPosition.Manual	Sowohl Größe als auch Position können manuell eingestellt werden.
FormStartPosition.CenterParent	Zentriert das Fenster in der Mitte des übergeordneten Fensters.
FormStartPosition.CenterScreen	Zentriert das Fenster in der Mitte des Desktops.
FormStartPosition.WindowsDefaultLocation	.net weist dem Fenster eine zufällige Position zu.
FormStartPosition.WindowsDefaultBounds	.net weist dem Fenster eine zufällige Position und eine zufällige Größe zu.

Tabelle 19: Größe und Position von Fenstern

Die Standardeinstellung ist dabei `WindowsDefaultLocation`, bei der die Position des Fensters von .net bestimmt wird.

Um ein Fenster beim Öffnen also beispielsweise in der Mitte des Desktops anzuzeigen, können Sie folgende Zeile verwenden.

```
MyForm.StartPosition = FormStartPosition.CenterScreen;
```

Bitte beachten Sie, dass Größen- und Positionsangaben im Konstruktor vernachlässigt werden, sofern Sie einen Wert für `StartPosition` verwenden, der eine bestimmte Größe oder Position impliziert.

20.6 Nützliche Eigenschaften

Neben den Eigenschaften, die Sie bisher zur Steuerung von Fenstern kennen gelernt haben, gibt es noch zahlreiche weitere. Einige von ihnen werden Sie nun noch kennen lernen.

Mittels der Eigenschaft `FormBorderStyle` können Sie festlegen, welchen Rahmen ein Fenster erhalten soll und ob es vergrößerbar ist oder nicht.

Als Parameter erwartet `FormBorderStyle` einen Wert der Enumeration `System.Windows.Forms.FormBorderStyle`

Wert	*Rahmenstil*	*Vergrößerbar*
FormBorderStyle.None	Weder Rahmen noch Titelleiste	Nein
FormBorderStyle.Sizable	Rahmen mit Titelleiste	Ja
FormBorderStyle.Fixed3D	3D-Rahmen mit Titelleiste	Nein
FormBorderStyle.FixedDialog	Dialogfenster mit Titelleiste	Nein
FormBorderStyle.FixedSingle	Wie **`FixedDialog`**	Nein
FormBorderStyle.SizableToolWindow	Kein Rahmen, schmale Titelleiste, ohne Systemmenü	Ja
FormBorderStyle.FixedToolWindow	Kein Rahmen, schmale Titelleiste, ohne Systemmenü	Nein

Tabelle 20: Rahmenparameter

Nachdem Sie mit `FormBorderStyle` nun festgelegt haben, über welchen Rahmen ein Fenster verfügt, können Sie noch bestimmen, welche Schaltflächen in der Titelleiste angezeigt werden sollen.

So macht es beispielsweise bei einem Dialogfenster, welches in der Größe nicht veränderbar ist, keinen Sinn, eine Schaltfläche zum Maximieren der Fenstergröße anzuzeigen.

Dazu können Sie die Eigenschaft `MaximizeBox` verwenden. Wenn Sie ihr das boolsche Literal `false` zuweisen, wird keine *Maximieren*-Schaltfläche in der Titelleiste angezeigt. Außerdem wird auch der zugehörige Eintrag im Systemmenü deaktiviert.

Ebenso können Sie mittels `MinimizeBox` festlegen, ob das Fenster über eine *Minimieren*-Schaltfläche verfügt. Die Eigenschaft `ControlBox` hingegen können Sie verwenden, um das komplette Systemmenü in der linken oberen Ecke eines Fensters zu entfernen.

Schließlich können Sie noch die Eigenschaft `ShowInTaskbar` verändern, um zu bestimmen, ob das Fenster als Eintrag in der Taskleiste erscheint oder nicht.

Ein übliches Dialogfenster, welches nicht vergrößerbar ist, nicht minimiert werden kann, über kein Systemmenü verfügt und auch nicht in der Taskleiste erscheint, können Sie beispielsweise mit den folgenden Zeilen initialisieren.

```
MyForm.FormBorderStyle = FormBorderStyle.FixedDialog;
MyForm.MaximizeBox = false;
MyForm.MinimizeBox = false;
MyForm.ControlBox = false;
MyForm.ShowInTaskbar = false;
```

20.7 Ausblick

Im nächsten Kapitel werden Sie die Grundlagen zum Zeichnen innerhalb von Fenster kennenlernen und erfahren, wie Sie Text in einem Fenster anzeigen können.

21 Zeichnen

In diesem Kapitel werden Sie die Grundlagen zum Zeichnen innerhalb von Fenster kennenlernen und erfahren, wie Sie Text in einem Fenster anzeigen können.

21.1 Einleitung

Für die Programme, welche Sie bisher für die Konsole geschrieben haben, gab es nur eine einzige Möglichkeit, Eingaben vom Benutzer entgegenzunehmen: Sie mussten die Methode `ReadLine` der Klasse `Console` aufrufen, um eine Zeichenkette von der Tastatur einzulesen.

In Programmen, die über eine grafische Benutzeroberfläche verfügen, wäre ein solches Vorgehen äußerst unpraktisch, da es neben der Tastatur beispielsweise noch die Maus als Eingabegerät gibt und der Benutzer zudem von anderen Programmen gewöhnt ist, dass er die Maus jederzeit benutzen kann.

Das bedeutet, dass man ständig eine Methode aufrufen müsste, die überprüft, ob der Benutzer mit der Maus irgendetwas angeklickt hat, um dann entsprechend reagieren zu können. Da ein solches Eingabemodell sehr aufwändig wäre, verfügen Programme in .net über einen anderen Ansatz: Ein ereignisgesteuertes Modell.

21.2 Ereignisse

Ein ereignisgesteuertes Modell bedeutet, dass die Anwendung etliche Methoden enthält, welche den Code enthalten, der ausgeführt werden soll, wenn die Maus beispielsweise bewegt wird oder mit ihr etwas angeklickt wird. Das Aufrufen dieser Methoden geschieht aber nicht innerhalb des Programms, sondern von außerhalb.

In der Tat überwacht .net alle Eingabegeräte und - falls mit einem dieser Geräte etwas passiert ist - sendet eine entsprechende Nachricht an das betroffene Programm und die passende Methode wird aufgerufen, sofern eine solche existiert. In dem Programm, das die Nachricht erhält, spricht man dann davon, dass ein Ereignis eingetreten ist.

Das bedeutet, dass ein Programm unter .net die meiste Zeit gar nicht mit der Ausführung von irgendwelchem Code verbringt, sondern hauptsächlich darauf wartet, dass das eine oder andere Ereignis eintritt. Genau dieses Warten erledigt die Methode `Application.Run`, die Sie bereits im letzten Kapitel kennengelernt haben.

Ereignisse werden in C# mittels der Schlüsselwörter `event` und `delegate` sowie mittels spezieller Methoden, so genannter Eventhandler unterstützt. Nähere Informationen zu diesem Thema können Sie noch einmal im Kapitel Klassenkonzepte[2] nachlesen.

21.3 Das Paint-Ereignis

Während die meisten Ereignisse unter .net dafür zuständig sind, Eingaben des Benutzers zu verarbeiten, gibt es einige Ereignisse, die eine besondere Rolle spielen. Eines dieser Ereignisse ist das `Paint`-Ereignis, das immer dann eintritt, wenn das Programmfenster neu gezeichnet werden muss.

Neu gezeichnet muss ein Fenster beispielsweise dann werden, wenn es geöffnet wird, wenn es minimiert war und nun wieder hergestellt wird, oder wenn es bisher von einem anderen Fenster verdeckt war und dieses Fenster verschoben oder geschlossen wird.

[2] Kapitel 16, Seite 134

Für das `Paint`-Ereignis gibt es einen speziellen Typ von Eventhandlern, nämlich den so genannten `PaintEventHandler`. Dieser wird innerhalb des .net Frameworks mit der folgenden Zeile deklariert.

```
public delegate void PaintEventHandler(object sender,
PaintEventArgs e)
```

Wie Sie wissen, ist ein Delegate eine Art Prototyp für eine Methode. Das heißt, dass eine Methode, welche auf das `Paint`-Ereignis reagieren soll, zwei Parameter erwartet, nämlich eines vom Typ `object` und ein zweites vom Typ `PaintEventArgs`.

Im ersten Parameter wird von .net das Objekt übergeben, welches das Ereignis ausgelöst hat. Dies ist beispielsweise dann notwendig, wenn mehr als ein Fenster zur gleichen Zeit angezeigt wird. Mit dem zweiten Parameter werden einige Optionen übergeben, mit denen das Neuzeichnen genauer gesteuert werden kann. Auf diese beiden Parameter wird später noch näher eingegangen.

21.4 Zeichnen

Um nun tatsächlich etwas zeichnen zu können, sind noch einige Schritte nötig. Zunächst muss eine entsprechende Methode angelegt werden, welche die eigentlichen Zeichenbefehle enthält. In dem folgenden Programm ist dafür die statische Methode `MyPaint` zuständig.

Diese Methode erhält über den Parameter `PaintEventArgs` eine Referenz auf ein Objekt vom Typ `Graphics`, welches seinerseits wiederum eine Referenz auf den Bereich eines Fensters enthält, der neu gezeichnet werden soll. Diese Referenz steckt innerhalb der `PaintEventArgs` in der Eigenschaft `Graphics`.

Mittels der Methode Clear wird der durch das Grafikobjekt beschriebene Bereich gelöscht und mit einer Hintergrundfarbe gefüllt. Diese Hintergrundfarbe muss als Objekt der Klasse Color übergeben werden. Diese Klasse, die Sie in einem späteren Kapitel noch genauer kennen lernen werden, dient zur Verwaltung von Farben.

Im Moment reicht es, zu wissen, dass die Klasse Color statische Attribute enthält, welche bestimmte Farben beschreiben, und die Sie - ohne sich näher um Farbdefinitionen kümmern zu müssen - einfach verwenden können, um bestimmte Standardfarben anzusprechen.

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

public class HalloWelt
{
  public static void Main()
  {
    Form form = new Form();
    form.Text = "Hallo Welt";
    form.Paint += new PaintEventHandler(MyPaint);

    Application.Run(form);
  }

  public static void MyPaint(object sender, PaintEventArgs e)
  {
    Graphics g = e.Graphics;
    g.Clear(Color.Blue);
  }
}
```

Code 55: Paint-Ereignis

Wenn Sie dieses Programm laufen lassen, wird sofort - nachdem das Hauptfenster geöffnet wurde - das Paint-Ereignis ausgelöst, da der Inhalt des Fensters neu gezeichnet werden muss. Da die selbstdefinierte Methode MyPaint zuvor mit dem Paint-Ereignis des Fensters verknüpft wurde, wird der Hintergrund des Fensters gelöscht und mit der Farbe blau überzeichnet.

21.5 Ungültige Bereiche

Wie Sie bereits gelernt haben, tritt das Paint-Ereignis nun jedes Mal ein, sobald das Fenster verschoben oder vergrößert wird, sobald es aus der Minimierung wiederhergestellt wird, sobald es nicht mehr von einem anderen Fenster verdeckt wird, ...

Allerdings wird von .net dabei nicht jedes Mal das komplette Fenster neu gezeichnet, da es oft ausreicht, nur einen Teil neu zu zeichnen. Dies sorgt auch dafür, dass für das Neuzeichnen nicht all zu viel Leistung benötigt wird.

Welcher Bereich eines Fensters tatsächlich neu gezeichnet wird, entscheidet .net intern, ohne dass der Anwender oder der Programmierer davon etwas mitbekommt. Das heißt insbesondere auch, dass man sich bei umfangreichen Zeichenfunktionen in der Methode MyPaint keine Sorgen wegen der Leistung des Programms machen muss, da .net in vielen Fällen entsprechende Optimierungen anwendet.

Der Bereich des Fensters, der schließlich neu gezeichnet wird, wird in .net als *ungültiger Bereich* bezeichnet. Man kann auch per Hand ein Fenster oder einen Teil eines Fensters als *ungültigen Bereich* definieren, um beispielsweise das Paint-Ereignis auszulösen. Wie das funktioniert, werden Sie später kennen lernen.

Um eine Vorstellung davon zu bekommen, wie oft das Paint-Ereignis beziehungsweise durch welche Aktionen es ausgelöst wird, können Sie die Methode MyPaint durch die Anweisung

```
Console.WriteLine("Das Paint-Ereignis ist eingetreten!");
```

erweitern, so dass Sie auf der Konsole jedes Mal eine entsprechende Meldung erhalten, wenn das Ereignis eingetreten ist.

Dazu ist es allerdings notwendig, dass Sie das Programm tatsächlich per Hand von der Konsole starten und nicht mittels einer grafischen Oberfläche, da sonst kein Konsolenfenster existiert, in welches die Meldung ausgegeben werden könnte.

21.6 Graphics

In der Methode `MyPaint` gibt es eine Anweisung, dis bisher nur sehr kurz erwähnt wurde, nämlich

```
Graphics g = e.Graphics;
```

Bisher wurde dazu nur gesagt, dass Sie über die Eigenschaft `Graphics` von `PaintEventArgs` eine Referenz auf den Bereich des Fensters erhalten, in den gezeichnet werden kann.

Diese Aussage soll nun etwas genauer betrachtet werden. Jedes Fenster besteht aus verschiedenen Bereichen, in die Sie teilweise mittels einer Methode wie `MyPaint` zeichnen können, teilweise aber auch nicht. Zu den Bereichen, die Sie nicht direkt verändern können, gehören beispielsweise die Titelleiste sowie der Rahmen eines Fensters.

Das Aussehen dieser Bereiche wird vom Betriebssystem vorgegeben und soll für alle Anwendungen gleich sein. Dass dieses Aussehen tatsächlich nicht von .net abhängt, lässt sich sehr gut unter Linux beobachten, wenn Sie beispielsweise den Windowmanager wechseln, die sich teilweise sehr deutlich voneinander unterscheiden.

Der Bereich, auf den Sie als Programmierer direkten Zugriff haben, ist der eigentliche Inhaltsbereich des Fensters und wird auch als `Client area` bezeichnet. Die Titelleiste und der Rahmen hingegen werden auch als `Non client area` bezeichnet.

Die Eigenschaft `Graphics` verschafft Ihnen nun eine Referenz auf genau diese *Client area*. Rein technisch gesehen liefert sie dazu ein Objekt der Klasse `System.Drawing.Graphics` zurück. Diese Klasse stellt Ihnen neben der entsprechenden Referenz noch zahlreiche Methoden zum Zeichnen von Grafiken oder Text zur Verfügung, wie beispielsweise die Methode `Clear`.

Die Klasse `Graphics` ist die Hauptmöglichkeit, auf die Grafikschnittstelle von .net zuzugreifen. Diese Grafikschnittstelle nennt sich *GDI*+, was als Abkürzung für *Graphics device interface* steht. GDI+ ist dabei eine so genannte *statuslose* Grafikschnittstelle, was bedeutet, dass einmal gezeichnete Objekte bei Änderungen am Fenster verloren gehen und nicht vom Grafiksystem zwischengespeichert werden. Daher muss nach jeder Fensteroperation der verloren gegangeneInhalt des Fensters neu gezeichnet werden.

21.7 Text ausgeben

Als nächstes werden Sie lernen, Text in einem Fenster auszugeben. Bisher kennen Sie zur Ausgabe von Text nur die Methode `Console.WriteLine`, welche Text aber nur auf die Konsole ausgeben kann. Dabei sind Sie immer auf eine Schriftart in einer festen Größe beschränkt, von verschiedenen Schriftvarianten wie *kursiv*, *fett* oder *unterstrichen* ganz zu schweigen.

Die Klasse `Graphics` stellt Ihnen daher eine Methode zur Verfügung, mit der Sie Schrift in allen möglichen Varianten innerhalb eines Fensters grafisch ausgeben können. Da die Schrift hierbei - wie auch alle anderen gezeichneten Objekte - als Grafik betrachtet wird, lässt sich auf ihr Aussehen viel mehr Einfluss nehmen als auf der Konsole.

Die entsprechende Methode heißt `DrawString`. Allerdings ist ihr Aufruf nicht ganz einfach, da sie etliche Parameter benötigt. Als erstes erwartet sie einen Ausdruck vom Typ `string`, welcher den eigentlich auszugebenden Text enthält. Beachten Sie, dass Sie diesen String wie schon auf der Konsole auch mit Variablen dynamisch zusammensetzen können.

Als zweiten und dritten Parameter werden zwei Objekte vom Typ `Font` und `Brush` erwartet, um die zu verwendende Schriftart sowie die Farbe festzulegen. Diese beiden Parameter werden Sie erst später detaillierter einsetzen, wenn die entsprechenden Klassen besprochen werden.

Für den Anfang reicht es aus, zu wissen, wie Sie eine Standardschrift und -farbe festlegen können. Beachten Sie, dass hier für die Farbe kein Objekt der Klasse `Color` verwendet wird, sondern eines der Klasse `Brush`. Dies liegt daran, dass mit diesem Parameter nicht nur eine einfache Farbe, sondern beispielsweise auch ein Farbverlauf oder ein Muster eingestellt werden kann.

Um diese beiden Objekte mit Standardwerten zu initialisieren, können Sie sich des ersten Parameters von `PaintEventArgs` bedienen - dem Objekt namens sender. Mittels der Anweisung

```
Form myForm = (Form)sender;
```

erhalten Sie eine Referenz auf das Fenster, welches neu gezeichnet wird. Mittels der Eigenschaft `Font` dieser Referenz können Sie die Standardschriftart eines Fensters ermitteln.

Manchmal bietet es sich an, diesen Wert zu Beginn der `MyPaint`-Methode einmalig zu ermitteln und dann in einer Variablen zu speichern, falls dieser Wert im weiteren Verlauf der Methode noch häufig benötigt wird.

Um eine Standardfarbe einzustellen, können Sie ähnlich wie bei Color vorgehen, nämlich sich einfach eines statischen Attributs der Klasse `Brushes` bedienen. In den ersten Beispielen können Sie dafür `Brushes.Black` verwenden, was eine schwarze Farbe einstellt, ohne weitere Muster oder Farbverläufe. Das heißt natürlich auch, dass Sie - so lange Sie noch keine weiteren Werte kennen - die Hintergrundfarbe des Fensters nicht auf schwarz stellen sollten.

Als letzte beiden Parameter erwartet die Methode `DrawString` noch zwei Werte vom Typ `float`, welche die x- und die y-Koordinate der linken oberen Ecke des auszugebenden Textes festlegen.

Das folgende Programm öffnet wiederum ein Fenster, setzt dessen Hintergrundfarbe auf weiß und schreibt in die linke, obere Ecke den Text *Hallo Welt!*.

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

public class HalloWelt
{
  public static void Main()
  {
    Form form = new Form();
    form.Text = "Hallo Welt";
    form.BackColor = Color.White;
    form.Paint += new PaintEventHandler(MyPaint);

    Application.Run(form);
  }

  public static void MyPaint(object sender, PaintEventArgs e)
  {
    Form myForm = (Form)sender;
    Graphics g = e.Graphics;

    g.DrawString("Hallo Welt!", myForm.Font, Brushes.Black, 0,
    0);
  }
}
```

Code 56: Weiße Hintergrundfarbe und schwarzer Text

Seien Sie vorsichtig, welche Anweisungen Sie in der Behandlung des `Paint`-Ereignisses verwenden! Benutzen Sie beispielsweise nie die Methode `MessageBox.Show` oder öffnen Sie weitere Fenster! Dabei könnte das ursprüngliche Fenster verdeckt werden, wodurch ein weiteres `Paint`-Ereignis ausgelöst würde, wodurch eine Endlosschleife entstehen könnte!

Verwenden Sie auch nie die Methode `Console.ReadLine`, da die Eingabe eventuell in einem Fenster, welches sich im Hintergrund befindet, erwartet wird. Sobald Sie dieses Fenster nach vorne holen, könnte unter Umständen ebenfalls ein weiteres `Paint`-Ereignis ausgelöst werden.

21.8 Vererbte Fenster

Bisher haben Sie nur Painteventhandler für Fenster geschrieben, welche innerhalb der `Main`-Methode angelegt worden sind. Wie Sie aber im letzten Kapitel gelernt haben, gibt es eine wesentlich komfortablere und einfachere Möglichkeit, Fenster anzulegen - nämlich mittels Vererbung.

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

public class HalloWelt : Form
{
  public HalloWelt()
  {
    this.Text = "Hallo Welt";
  }

  public static void Main()
  {
    Application.Run(new HalloWelt());
  }
}
```

Code 57: Vererbung von Fenstern

Das Problem ist hierbei nun, dass es keine Referenz auf das Fenster gibt, welche Sie verwenden könnten, um dem Ereignis `Paint` mittels des `+=`-Operators einen neuen Eventhandler anzuhängen. Die Frage ist also, wie sich die Verknüpfung in diesem Fall herstellen lässt.

Für diesen Fall existiert eine virtuelle Methode namens `OnPaint` in der Klasse `Form`, die Sie in Ihrer eigenen Klasse überschreiben können. Da bei einem Aufruf von `OnPaint` klar ist, aus welchem Fenster das Ereignis stammt, kann bei den Parametern das Objekt `sender` entfallen, so dass nur noch ein einzelner Parameter vom Typ `PaintEventArgs` benötigt wird.

Beachten Sie, dass diese virtuelle Methode nicht nur in der Klasse `Form` existiert, sondern auch in allen anderen Steuerelementen wie beispielsweise Schaltflächen oder Listen, die Sie später noch kennen lernen werden.

Das folgende Beispiel veranschaulicht die Verwendung der Methode `OnPaint`. Da der Aufruf von `OnPaint` nicht zusätzlich zu der `OnPaint`-Methode der Basisklasse stattfindet, sondern diese ersetzt, sollte man als erste Anweisung die entsprechende Methode der Basisklasse aufrufen.

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

public class HalloWelt : Form
{
  public HalloWelt()
  {
    this.Text = "Hallo Welt";
    this.BackColor = Color.White;
  }

  protected override void OnPaint(PaintEventArgs e)
  {
    base.OnPaint(e);

    Graphics g = e.Graphics;
    g.DrawString("Hallo Welt!", this.Font, Brushes.Black, 0,
    0);
  }

  public static void Main()
  {
    Application.Run(new HalloWelt());
  }
}
```
Code 58: Verwenudng der Methode OnPaint