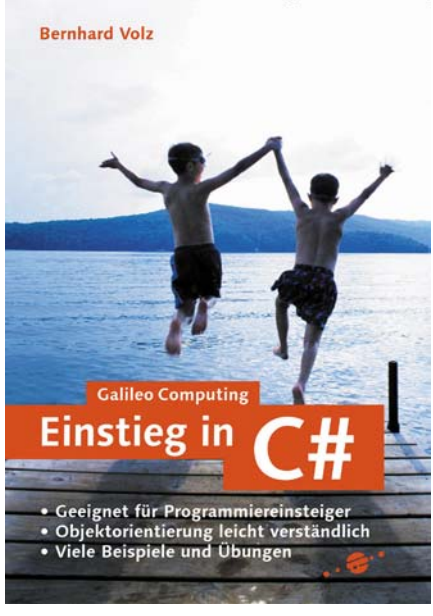
Bernhard Volz

Einstieg in C#



Galileo Computing

Inhalt

1 Einführung 15

1.1 .NET 15
1.1.1 Verwendung mehrerer Programmiersprachen für ein Projekt 18
1.1.2 Garbage Collection und Sicherheit 20

1.2 C# 21

1.3 Zielgruppe 22

1.4 Danksagungen 22

1.5 Kontakt 23

1.6 Aufbau des Buches 23

1.7 Die Buch-CD 24

1.8 Einrichten einer Entwicklungsumgebung 25
1.8.1 Installation von .NET 25
1.8.2 Installation von SharpDevelop 26
1.8.3 Installation von Visual Studio .NET 26
1.8.4 Hinweise zur Kommandozeile 27

Teil I Erste Schritte 29

2 Aufbau von C#-Programmen 31

2.1 Hello, World! 31

2.2 Kommentare 33
2.2.1 Kommentarblöcke 34
2.2.2 Zeilenkommentare 35

2.3 Syntax und Semantik 36

2.4 Verwendete Syntax-Schreibweise 37

2.5 Eine kurze Einführung zum Thema »Klassen« 37
2.5.1 Deklarieren von Klassen 38
2.5.2 Der Einsprungspunkt 38

2.6 Zusammenhang zwischen Klassen- & Datei-Name 39

2.7 Ausgaben auf dem Bildschirm 40
2.7.1 Namensräume 40
2.7.2 Umbruch der Ausgabe von WriteLine() 41

2.8 Assembly 41

2.9 Zusammenfassung 43
2.9.1 Bestandteile eines C#-Programms 44
2.9.2 Zusätzliches über den .NET-Framework 44
2.9.3 C#-Sprachelemente 45

2.10 Übungen 45

3 Konstanten, Variablen & Datentypen 47

3.1 Das EVA-Prinzip 47

3.2 Variablen 48
3.2.1 Primitive Datentypen in C# 49
3.2.2 Datentypen für ganzzahlige Werte (Menge der ganzen Zahlen) 50
3.2.3 Datentypen für gebrochene Werte (Menge der rationalen Zahlen) 51
3.2.4 Wahrheitswerte 53
3.2.5 Zeichenketten 53
3.2.6 Namenskonventionen für Variablen 54
3.2.7 Deklaration von Variablen 56
3.2.8 Wertzuweisung und Initialisierung von Variablen 57
3.2.9 Ausgabe von Variableninhalten 63
3.2.10 Gültigkeit von Variablen 68

3.3 Felder 70
3.3.1 Deklaration und Initialisierung 71
3.3.2 Zugriff und Ausgabe von Arrays: [..], foreach 72
3.3.3 Mehrdimensionale Arrays 73
3.3.4 Unregelmäßige Arrays 75
3.3.5 Speicherbereinigung 75

3.4 Parameter der Main-Funktion 76

3.5 Typqualifizierer 77
3.5.1 static 78
3.5.2 const 78

3.6 Einlesen von Variablenwerten über die Tastatur 78

3.7 Zusammenfassung 80

3.8 Übungen 81

4 Zusammengesetzte Datentypen und Namensräume 83

4.1 Strukturen 83
4.1.1 Deklaration einer Struktur 83
4.1.2 Eine Struktur zur Aufnahme von Adressen 87
4.1.3 Werte- und Verweistypen 88
4.1.4 Boxing und Unboxing 91

4.2 Aufzählungen (Enumerationen) 92
4.2.1 Deklaration einer Aufzählung 93
4.2.2 Enumerationen und Zahlwerte 94
4.2.3 Basistyp einer Enumeration 95

4.3 Namensräume 96
4.3.1 Definition eines Namensraums 97
4.3.2 Die using-Klausel 99

4.4 Zusammenfassung 100
4.4.1 Strukturen 100

4.4.2 Enumerationen 101
4.4.3 Namespaces 101

4.5 Übungen 102

5 Operatoren 103

5.1 Operatoren in C# 103

5.2 Additive und multiplikative Operatoren 105
5.2.1 Addition, Subtraktion, Multiplikation und Division 105
5.2.2 Division mit Rest (Division modulo x) 107

5.3 Der Zuweisungsoperator 107

5.4 Primäre Operatoren 108
5.4.1 Klammerung »()« 108
5.4.2 Memberzugriff "." 109
5.4.3 Methodenaufruf 110
5.4.4 Array-Zugriff »[ ]« 110
5.4.5 Post-Inkrement und Post-Dekrement 110
5.4.6 Anlegen von Objekten – der new-Operator 112
5.4.7 Typ und Größe einer Variablen 113
5.4.8 Geprüfte und ungeprüfte Ausführung von Operationen 115

5.5 Unäre Operatoren 117
5.5.1 Vorzeichen 117
5.5.2 Negationen 118
5.5.3 Pre-Inkrement und Pre-Dekrement 119
5.5.4 Typumwandlung 119

5.6 Schiebe-Operatoren 121

5.7 Relationale und Vergleichsoperatoren 122
5.7.1 Vergleichsoperatoren 122
5.7.2 Die Operatoren is und as 123

5.8 Logisches UND, ODER und EXKLUSIV-ODER (XOR) 124

5.9 Bedingtes UND und ODER 125

5.10 Bedingung 125

5.11 Zusammenfassung 126

5.12 Übungen 127

6 Kontrollkonstrukte 129

6.1 Nassi-Shneiderman 129

6.2 Schleifen 130
6.2.1 for 131
6.2.2 while 135
6.2.3 do-while 138
6.2.4 foreach 141
6.2.5 Steuerung der Schleifenabläufe: break & continue 141

6.3 Bedingungen (bedingte Anweisungen) 142
6.3.1 if 142
6.3.2 switch-case 148
6.4 goto 151
6.5 Zusammenfassung 152
6.6 Übungen 153

Teil II Objektorientierte Programmierung mit C# 157

7 Einführung in die Objektorientierte Programmierung 159

7.1 Klassen und Objekte 159
7.1.1 Die Klasse als programmiersprachliche Beschreibung von realen Objekten 159
7.1.2 Objektmethoden 167
7.1.3 Eine spezielle Methode: Der Konstruktor 178
7.1.4 Die Vererbungslehre in der Programmiersprache 183
7.1.5 Gleichheit von Objekten 195
7.1.6 Überladen von Methoden 196
7.1.7 Überschreiben von Methoden: virtuelle Methoden 199
7.2 Abstrakte Klassen 202
7.2.1 Abstrakte Methoden 203
7.3 Eigenschaften (Properties) 205
7.3.1 get 206
7.3.2 set 207
7.3.3 Benutzung von Properties 208
7.4 Schnittstellen 209
7.4.1 Deklaration einer Schnittstelle 210
7.4.2 Implementierung einer Schnittstelle 211
7.5 Klassenmember (statische Member) 215
7.5.1 Statische Methoden 216
7.5.2 Statische Daten 216
7.5.3 Statische Daten in nicht statischen Methoden 217
7.5.4 Erzeugung von Objekten mit privaten Konstruktoren 217
7.6 Ref- & Out-Parameter von Methoden 218
7.6.1 Ref-Parameter 218
7.6.2 Out-Parameter 220
7.7 Die Speicherverwaltung von .NET 221
7.7.1 Finalize() und C#-Destruktoren 222
7.7.2 Referenzen auf Objekte während Finalisierung 224
7.7.3 Dispose(): Eine bessere Lösung als Finalize() 224
7.8 Objektorientierte Schmankerl in C# 226
7.8.1 Neuimplementierung einer Methode 226
7.8.2 sealed vor virtuellen Methoden 231
7.8.3 Zugriff auf Implementierung der Basisklasse: base Teil II 231

7.9 Weitere Elemente der Unified Modelling Language 234
7.9.1 Schnittstellen 234
7.9.2 Assoziationen 234
7.10 Zusammenfassung 235
7.10.1 Klassen 235
7.10.2 Schnittstellen 236
7.10.3 Properties 236
7.10.4 Methoden 237
7.11 Übungen 237

8 Strings & reguläre Ausdrücke 241

8.1 Zeichenketten 241
8.1.1 string vs. String 241
8.1.2 Länge von Zeichenketten 242
8.1.3 Iteration über eine Zeichenkette 243
8.1.4 Vergleich zweier Zeichenketten 244
8.1.5 Untersuchung von Zeichenketten 247
8.1.6 Splitting von Strings 248
8.1.7 Zurechtschneiden von Zeichenketten 249
8.1.8 Groß- und Kleinschreibung 250
8.1.9 Löschen und Ersetzen von Zeichen 250
8.1.10 Einfügen von Zeichen 251
8.1.11 Teilstrings herauslösen 251
8.1.12 Weitere Operationen mit Strings im Überblick 252
8.1.13 Format() 252
8.2 Dynamische Zeichenketten – StringBuilder 253
8.2.1 Instanziierung eines StringBuilder-Objekts 253
8.2.2 Eigenschaften und Methoden 253
8.2.3 ToString() 255
8.3 Reguläre Ausdrücke 256
8.3.1 Regex: Eine Klasse für reguläre Ausdrücke 257
8.3.2 Grundlegender Aufbau eines regulären Ausdrucks 257
8.3.3 Optionen 260
8.4 Zusammenfassung 264
8.5 Übungen 265

9 Ausnahmen (Exceptions) 267

9.1 Der klassische Ansatz: Rückgabewerte 267
9.2 Exception-Mechanismus 269
9.2.1 Exceptions – Allgemein 269
9.2.2 Exceptions – Wie funktioniert's? 270
9.2.3 Exceptions – Nachteile 270
9.2.4 Exceptions und .NET 271

9.3 Exceptions in C# 272
9.3.1 Das Werfen einer Exception 272
9.3.2 Das Fangen einer Exception 272
9.3.3 Member der Klasse Exception 274
9.3.4 Exception-Klassen 275
9.3.5 Eigene Exception-Klassen 278
9.3.6 Aufräumarbeiten: finally 279
9.3.7 Verhaltensweisen beim Auftreten einer Ausnahme 280
9.4 Zusammenfassung 283
9.5 Übungen 284

10 Überladen von Operatoren 287
10.1 Unäre Operatoren 288
10.2 Binäre Operatoren 290
10.3 Vergleichsoperatoren 291
10.3.1 Equals() 292
10.4 Einschränkungen 293
10.5 Zusammenfassung 293
10.6 Übungen 293

11 Delegates und Ereignisse 295
11.1 Delegates 295
11.1.1 Beispiel: Motorüberwachung 295
11.1.2 Die Deklaration eines Delegates 296
11.1.3 Die Verwendung von Delegates 297
11.1.4 Erstellen eines Delegates 299
11.1.5 Multicast Delegates 302
11.1.6 Callback-Methoden und Ausnahmen 305
11.1.7 Delegates und Rückgabewerte 306
11.2 Ereignisse 307
11.2.1 Delegate vs. Ereignis 307
11.2.2 Ereignisse: Hintergrund 310
11.2.3 Deklaration eines Ereignisses 310
11.2.4 Hinweise 312
11.2.5 Ereignisse und Rückgabewerte 314
11.3 Zusammenfassung 314
11.4 Übungen 315

12 Indizierer, Enumeratoren und Collections 317
12.1 Indizierer 317
12.1.1 Allgemeines über Indizierer 317
12.1.2 Deklaration eines Indizierers 318

12.1.3 Die Verwendung eines Indizierers 320
12.1.4 Indizierer mit mehreren Parametern 323

12.2 Enumeratoren 323
12.2.1 IEnumerable und IEnumerator 324
12.2.2 Ein Enumerator für BitVector64 326
12.2.3 Erweiterungen für die Unterstützung des Enumerators an BitVector64 329
12.2.4 Verwendung des Enumerators 330
12.2.5 Zur Perfektion fehlt noch etwas 331
12.2.6 Noch einmal foreach 333

12.3 Collections in .NET 334
12.3.1 ArrayList 334
12.3.2 Queue 337
12.3.3 Stack 339

12.4 Zusammenfassung 341

12.5 Übungen 342

13 Attribute und Metadaten 343

13.1 Attribute 343
13.1.1 Attribute im Code platzieren 344
13.1.2 Reservierte Attribute 345
13.1.3 Eigene Attribute entwickeln 351
13.1.4 Ziele für Attribute 355

13.2 Metadaten 355
13.2.1 Typinformationen zur Laufzeit ermitteln 356
13.2.2 Attribute auslesen 365
13.2.3 Ausblick 367

13.3 Zusammenfassung 367

13.4 Übungen 368

14 XML-Dokumentation und Präprozessor 369

14.1 XML-Dokumentation 369
14.1.1 XML als Datenformat 369
14.1.2 Eine weitere Form des Kommentars: /// 374
14.1.3 Dokumentationstags 375
14.1.4 Textauszeichnung und Verweise 381
14.1.5 NDoc 385

14.2 Präprozessor 385
14.2.1 Ein- und Ausblenden von Code 385
14.2.2 Weitere Direktiven 388

14.3 Zusammenfassung 390

14.4 Übungen 390

15 Threading 393

15.1 Betriebssystem-Hintergrund: Prozesse und Threads 393
15.1.1 Programme: Historie 393
15.1.2 Prozesse und Threads 395
15.1.3 Parallelität durch den Einsatz von Threads 396

15.2 Threads in C# 397
15.2.1 Thread-Erzeugung 397
15.2.2 Auch Threads brauchen ihren Schlaf 401
15.2.3 Suspendieren von außen 406
15.2.4 Abbruch eines Threads 408
15.2.5 Warten auf das Ende eines Threads 410
15.2.6 Aktueller Thread-Zustand 410
15.2.7 Thread-Prioritäten 411
15.2.8 Name eines Threads 412
15.2.9 Zusammenfassung der wichtigsten Methoden und Properties der Klasse Thread 413
15.2.10 Übergabe und Rückgabe von Daten 414

15.3 Synchronisierung 414
15.3.1 Die Gefahr von nicht synchronisierten Threads 415
15.3.2 Die Klasse Monitor 417
15.3.3 Ein Monitor in C# 419

15.4 Asynchrone Methodenaufrufe 420
15.4.1 BeginInvoke() und EndInvoke() 421
15.4.2 Methodenaufruf ohne Parameter und Rückgabewert 421
15.4.3 Methodenaufruf mit Parametern und Rückgabewert 424

15.5 Zusammenfassung 427

15.6 Übungen 428

Teil III Weiterführende Themen zur Programmierung unter .NET 431

16 Einführung in Windows Forms 433

16.1 Das WinForms-Anwendermodell 433
16.1.1 Der Form-Begriff 433
16.1.2 Steuerelemente (Controls) 434
16.1.3 Ereignisse 434
16.1.4 Zerstörung von Elementen 435
16.1.5 Ein einfaches Beispiel 436

16.2 Fenster-Layout 437

16.3 Controls 441
16.3.1 Reaktion auf die Betätigung einer Schaltfläche 441
16.3.2 Übersicht über die wichtigsten Controls 442

16.4 Menüs 443

16.5 Dialoge 447

16.6 Zeichnen in Fenstern 452
16.6.1 Koordinaten in Windows-Fenstern 457
16.6.2 Farben 458
16.6.3 Zeichenmethoden des Graphics-Objekts 459

16.7 Zusammenfassung 463

16.8 Übungen 464

17 Bibliotheken und CodeDOM 467

17.1 Bibliotheken 467
17.1.1 Statische und dynamische Bibliotheken 467
17.1.2 Unterschied zwischen einem Programm und einer Bibliothek 468
17.1.3 Ein einfaches Beispiel 468
17.1.4 Projekte mit mehreren Code-Dateien 470

17.2 CodeDOM 471
17.2.1 Ein Simulator für ein Makro 472
17.2.2 Die Oberfläche der Anwendung 473
17.2.3 Code zur Laufzeit erzeugen 473
17.2.4 Vorbemerkungen 474
17.2.5 Die Methode OnClickedStart() 474

17.3 Zusammenfassung 478

17.4 Übungen 479

A Visual Studio .NET 481

A.1 Erste Schritte 481

A.2 Das Erstellen eines C#-Projekts 484

A.3 Fehlersuche in Programmen: Debuggen 490

A.4 Beenden von Visual Studio .NET 497

A.5 Fazit 497

B SharpDevelop 499

B.1 Erste Schritte 499

B.2 Übersetzen und Ausführen von Projekten 506

B.3 Debuggen 506

B.4 Fazit 507

C Der Microsoft CLR-Debugger 509

C.1 Start 509

C.2 Debuggen eines Programms 510

C.3 Beenden des Debugging 515
C.4 Fazit 515

D WinCV & QuickStart Tutorials 517
D.1 WinCV 517
D.2 QuickStart Tutorials 518

2 Aufbau von C#-Programmen

In diesem Kapitel lernen Sie den grundlegenden Aufbau von C#-Programmen kennen. Darüber hinaus werde ich Sie mit Begriffen wie »Klasse«, »Objekt«, »namespace« und »Assembly« sowie Syntax und Semantik einer Programmiersprache vertraut machen.

Bevor wir uns in eine Unmenge von neuen Begriffen stürzen, möchte ich Ihnen gerne ein erstes, kleines Programm – in C# geschrieben – vorstellen. Es trägt den Namen »Hello-World«, benannt nach der Ausgabe, die es am Bildschirm erzeugt.

Dieses Programm besitzt einen großen Bekanntheitsgrad in der Entwicklergemeinde: Fast jeder Programmierer hat es mindestens einmal geschrieben. Es veranschaulicht auf einfache Art und Weise, aus welchen Sprachelementen ein minimales Programm besteht. Seit seiner »Erfindung« vor vielen Jahren hat es sich hartnäckig in der einschlägigen Literatur gehalten.

2.1 Hello, World!

Da ich mit dieser »Tradition« nicht brechen möchte, finden Sie im nachfolgenden Listing 2.1 die C#-Version[1] des »Hello-World«-Programms.

```
01: /*
02:   HelloWorld.cs
03:   Das Programm gibt "Hello, World!" am Bildschirm aus
04:   und beendet sich danach
05: */
06:
07: class HelloWorld
08: {
09:    public static void Main()
10:    {
11:      System.Console.WriteLine("Hello, World!");
12:    }
13: }
```

Listing 2.1 Das erste Programm in einer neuen Programmiersprache

1 Im Buch »C#« von Eric Gunnerson (ebenfalls bei Galileo Computing erschienen), heißt es auch »Hello, Universe!« – wiederum frei nach dem ausgegebenen Text.

Schreiben Sie das Programm einfach wie in Kapitel 1 beschrieben mit einem Texteditor (ohne die Zeilennummern) ab und rufen Sie danach auf der Kommandozeile den C#-Compiler auf. Ich habe als Namen für dieses Programm **HelloWorld.cs**[2] gewählt. Alle Dateien, die C#-Code enthalten, sollten mit der Erweiterung **cs** benannt werden.

Auch wenn das Kopieren von CD der hier vorgestellten Programme sehr einfach geht, schreiben Sie sie ab! Viele Dinge lernt man besser, wenn man sie praktisch übt. Dazu gehören auch simpel aussehende Programme.

Wenn Sie alles richtig abgetippt haben, können Sie das Programm reibungslos mit dem Compiler übersetzen (Listing 2.1).

Der Aufruf des Übersetzers erfolgt durch den Befehl `csc`, dem als Parameter der Name der zu übersetzenden Datei (inklusive Dateinamenerweiterung) übergeben werden muss. Um das Hello-World-Programm zu übersetzen, muss somit `csc HelloWorld.cs` auf der Kommandozeile eingegeben werden.

Sollten im Programm Fehler vorhanden sein, meldet Ihnen diese der Übersetzer mit einer (mehr oder weniger genauen) Fehlermeldung sowie der betreffenden Zeilennummer.

C#-Programme sind case-sensitive. D.h., es wird zwischen Groß- und Kleinschreibung unterschieden! Wenn Fehler auftreten, vergewissern Sie sich zunächst, dass nicht nur das entsprechende Wort korrekt geschrieben wurde, sondern auch die Schreibweise zu 100 % übereinstimmt!

Machen Sie sich frühestmöglich mit einfachen, schnell unterlaufenen Fehlern wie vergessenen Zeichen[3] vertraut – d.h., welche Fehlermeldungen werden in diesen Fällen erzeugt. Und vor allem: Schrecken Sie nicht vor vielen Fehlermeldungen zurück! Oftmals verursacht ein Fehler etliche weitere und die Beseitigung des ersten Fehlers »behebt« die nachfolgenden von allein.

2 Sie finden die Quelltexte zu den hier vorgestellten Programmen auch auf der Begleit-CD. Wechseln Sie hierzu in das Verzeichnis »SourceCode« auf der CD; anhand von Kapitelnummer und hier angegebenen Dateinamen finden Sie dann den Quellcode in den entsprechenden Unterverzeichnissen.

3 Ein Paradebeispiel für ein solches Zeichen ist das Semikolon – jede Anweisung in C# muss mit einem solchen Zeichen abgeschlossen werden.

Sind keine Fehler in der Eingabedatei vorhanden, erzeugt der Compiler eine ausführbare Datei – erkennbar an der Datei-Erweiterung exe. Diese können Sie nun durch Eingabe des Dateinamens starten.

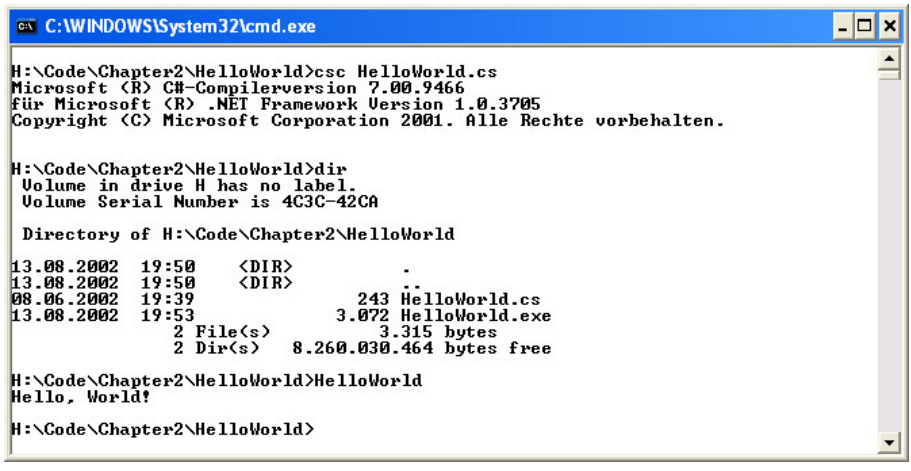


Abbildung 2.1 Ein Aufruf des C#-Compilers erzeugt eine neue, ausführbare Datei.

Wenn Sie diese Anwendung, wie in Abbildung 2.1 zu sehen ist, durch die Eingabe des Dateinamens `HelloWorld` starten, wird der Text »Hello, World!« auf den Bildschirm geschrieben. Wenn Sie all diese Schritte erfolgreich hinter sich gebracht haben, d. h., Sie sehen die Ausgabe des Programms, dann haben Sie soeben Ihr erstes Programm in C# geschrieben!

In den folgenden Abschnitten werden wir nun dieses Programm in seine einzelnen Bestandteile zerlegen und eingehend betrachten.

2.2 Kommentare

Wie im Listing 2.1 in den ersten Zeilen auffällt, scheint der Text zwischen den Zeichen /* und */ scheinbar nichts mit dem Programm zu tun zu haben: Es handelt sich hierbei um einen Kommentar des Programmautors.

Kommentare werden beim Übersetzen vom Compiler nicht weiter beachtet. Sie dienen dazu, Anmerkungen zum Programm im Code zu machen, die zu einem besseren Verständnis des Quellcodes beitragen sollen.

Man setzt für gewöhnlich an den Anfang einer Datei zunächst immer einen Kommentar, der den Inhalt des Files beschreibt. Auch Angaben zu Autor, Version, Copyright und Änderungen an der Datei sind Thema dieses Kommentars. Hier-

durch erhält man einen besseren Überblick über die Historie einer Datei (z.B. warum eine bestimmte Änderung vorgenommen wurde). Insbesondere bei größeren Projekten, an denen mehr als zwei Leute arbeiten, ist dies eine große Hilfe!

```
01: /*
02:    HelloWorld.cs
03:   ...
04: */
```

Listing 2.2 Zu jedem Programm gehören erläuternde Kommentare am Anfang einer Datei

Kommentare können aber auch dazu eingesetzt werden, komplizierte Algorithmen zu erläutern. Dies erleichtert die Arbeit von nachfolgenden Entwicklern ungemein, die diesen Code-Abschnitt verstehen müssen (z.B. um ihn zu erweitern).

2.2.1 Kommentarblöcke

Ein Kommentar-Abschnitt, d.h. eine Anmerkung, die sich über einen festgelegten Bereich erstreckt, wird wie in Listing 2.2 gezeigt, durch die Zeichenfolge `/*` eingeleitet und durch die Zeichen `*/` abgeschlossen. Zeilenumbrüche dazwischen werden ignoriert; d.h., der Kommentar kann sich über mehrere Zeilen hinweg erstrecken, muss es aber nicht!

Ein weiterer Einsatzzweck dieser Art der Kommentierung ist das Ausblenden von Code-Abschnitten, ohne diese physikalisch – d.h. aus der Datei – zu entfernen. Man setzt Beginn- und Ende-Zeichen an die entsprechenden Code-Abschnitte (Listing 2.3) und verhindert dadurch die Einbeziehung des so umschlossenen Code-Abschnitts in den Übersetzungsvorgang.

```
01: ...
02: public static void Main()
03: {
04:    /* System.Console.WriteLine("ausgeblendet"); */
05:    System.Console.WriteLine("aktiv");
06: }
07: ...
```

Listing 2.3 Auskommentieren von Code verhindert die Einbindung in den Übersetzungsvorgang

Dieses Vorgehen findet Anwendung, wenn man neuen Code einfügt und den bestehenden dabei so verändert, dass er nicht restauriert werden kann – also für Sicherungszwecke. Aber auch wenn »mal schnell« eine andere Variante des Pro-

grammcodes getestet werden soll[4], greift man auf die Auskommentierung[5] zurück.

Eine Schachtelung von Kommentaren, wie in Listing 2.4 dargestellt, ist nicht möglich. Der innere Kommentar beendet automatisch alle äußeren – im Beispiel beenden die Ende-Zeichen von Kommentar 2 automatisch den Kommentar-Block 1. Auch eine Schachtelung über Kreuz, d.h., Kommentar 2 wird außerhalb von Kommentar 1 beendet, ist nicht möglich!

```
01: /* Beginn Kommentar 1
02:    /* Beginn Kommentar 2
03:    Ende Kommentar 2 */
04: Ende Kommentar 1 */
```

Listing 2.4 Die Schachtelung von Kommentar-Blöcken erzeugt beim Übersetzen einen Fehler

Achten Sie daher bei Verwendung dieser Methode darauf, dass keine Überschneidung von Kommentaren auftritt! Entwicklungswerkzeuge unterstützen Sie dabei zumeist durch Darstellung der Kommentare in einer anderen Farbe – so kann sehr schnell erkannt werden, was noch vom Compiler als Kommentar gewertet wird.

2.2.2 Zeilenkommentare

Neben diesem Abschnittskommentar gibt es eine weitere Art, den *Zeilenkommentar*. Er stammt im Gegensatz zu der bereits vorgestellten Art nicht von C ab, sondern von C++. Die Zeichenfolge // leitet einen Zeilenkommentar für die aktuelle Zeile ab Beginn der Zeichen ein (Listing 2.5).

```
01: ...
02: System.Console.WriteLine("Hello, World!"); // Ausgabe
03: ...
```

Listing 2.5 Zeilenkommentare haben kein explizites Ende-Zeichen

Der Kommentar endet automatisch mit dem Ende der betreffenden Zeile; d.h., ein Zeilenumbruch impliziert das Ende des Kommentars – im Gegensatz zum expliziten Ende der vorhergehenden Variante.

Zeilen- und Abschnittskommentare können gemischt werden (Listing 2.6)!

4 Das so genannte »try-and-error principle« erfreut sich besonders unter Anfängern in Sachen Programmierung einer sehr großen Beliebtheit.

5 Auskommentieren sollte aber nicht dazu verwendet werden, um alten Code permanent zu konservieren. Stattdessen sollte man auf ein Versionierungssystem zurückgreifen, das erlaubt, alte Versionen zu sichern und auch wiederherzustellen.

```
01: /*
02:     System.Console.WriteLine("Hello, World!") // Ausgabe
03: */
```

Listing 2.6 Das Mischen von Zeilen- und Kommentarblöcken ist möglich

Der Zeilenkommentar kommt in C# noch in einer weiteren Spezialform vor: Diese kann zur automatischen Generierung von Programm-Dokumentation genutzt werden. Im Kapitel 14 finden Sie hierzu weitere Informationen.

Kommentare sollten kurz und prägnant ausfallen! Es gibt »schlecht« und »gut« kommentierte Programme: Zu schlecht kommentierten zählt man neben denen, die gar keine Kommentare enthalten, auch diejenigen, die zu viele beinhalten oder nur das Offensichtliche[6] dokumentieren.

Selten aber trifft man auf das Extrem, dass ein Programm mit zu vielen Kommentaren versehen wurde. Der Grund hierfür ist die Tatsache, dass das Kommentieren zu den unbeliebtesten Arbeiten beim Schreiben von Software gehört. Oft trifft man daher das andere Extrem an – zu wenig oder auch unverständliche Kommentare.

2.3 Syntax und Semantik

Unter der **Syntax** einer Programmiersprache versteht man umgangssprachlich: »Wie wird etwas ausgedrückt«. Syntax-Definitionen beschreiben also, wie Konstrukte aufgebaut sind und welche Variationsmöglichkeiten man hat.

Im Gegensatz zur Syntax, die das »Wie« beschreibt, versteht man unter der **Semantik** die Bedeutung des tatsächlich geschriebenen Ausdrucks.

Nehmen wir als Beispiel einen einfachen Ausdruck: »4-3«. Die Syntax beschreibt nun, wie eine Subtraktion formal niedergeschrieben wird. Vertauscht man die beiden Ziffern im Ausdruck zu »3-4«, ändert sich nicht die Syntax; es ist immer noch eine gültige Subtraktion. Stattdessen ändert sich die Semantik: Das Ergebnis aus dem ersten Fall ist nicht gleich dem zweiten Ergebnis, bezogen auf die numerischen Werte.

Im weiteren Verlauf dieses Buches werde ich Ihnen immer wieder Syntaxdefinitionen für die unterschiedlichsten Ausdrücke und Konstrukte von C# vorstellen.

6 Ist aus dem Code ersichtlich, dass z. B. der Wert einer Variablen um eins erhöht wird, so ist dies nicht durch einen Kommentar zu erläutern – diese Zeile sollte jeder, der dieser Sprache mächtig ist, ohne weitere Erläuterungen verstehen.

2.4 Verwendete Syntax-Schreibweise

In diesem Buch werden sehr oft Ausdrücke in spitzen Klammern auftauchen (z. B. `<Klassenname>`). Diese Schreibweise werde ich dann verwenden, wenn ich eine Syntaxdefinition beschreiben möchte. Die spitzen Klammern sind kein Teil der C#-Syntax und müssen nicht mit abgeschrieben werden. Vielmehr stehen Sie als eine Art Platzhalter für ein Stück Code bzw. Angaben des Programmierers.

Ähnlich den spitzen Klammern werde ich eckige Klammern dazu einsetzen, optionale Elemente zu beschreiben. Diese Elemente können Sie mit angeben, müssen dies aber nicht zwingend.

Ich beschränke mich auf die Verwendung dieser beiden Zeichenpaare. Falls diese Teil der Syntax sind, d. h. mit im Code geschrieben werden müssen, werde ich dies bei jeder Syntaxdefinition entsprechend im Text vermerken.

2.5 Eine kurze Einführung zum Thema »Klassen«

Obwohl das Thema Objekte & Klassen in den zweiten Teil dieses Buches gehört, möchte ich an dieser Stelle kurz darauf eingehen. Der Grund ist, dass C# immer mindestens eine Klasse benötigt, die den **Einsprungspunkt**[7] für das Programm enthält.

Unter einer **Klasse** versteht man ein abstraktes Gebilde, das ein real existierendes »Objekt« nachbildet. Es vereint dem Objekt zugehörige Daten und Methoden (Aktionen, die das Objekt ausführen kann). Eine Klasse bezeichnet man auch als Gesamtheit eines Objekts.

Ein **Objekt** – im programmiertechnischen Sinn – ist eine **Instanz** einer Klasse, d. h. eine zu Nullen und Einsen gewordene Abbildung im Speicher des Computers, die nach der Vorlage der Klasse erzeugt wurde.

Diese beiden Definitionen erheben nicht den Anspruch, vollständig zu sein. Für den jetzigen Zeitpunkt sind sie aber bei weitem ausreichend. Zu gegebener Zeit werde ich sie ergänzen.

Wenn Sie Sekundärliteratur wie Dokumentationen aus dem Internet einsetzen oder sich mit Entwicklern unterhalten, werden Sie sicher schnell feststellen, dass die Begriffe **Klasse** und **Objekt** häufig durcheinander gebracht werden. Lassen Sie sich daher nicht davon beirren! Im Folgenden gilt immer die oben angeführte Definition!

7 Jedes Programm enthält einen so genannten *Einsprungspunkt*. Er kennzeichnet die Stelle im Code, die nach dem Start der Anwendung angesprungen wird, d. h. den Ort, an dem die eigentliche Ausführung des Programms beginnt.

2.5.1 Deklarieren von Klassen

Eine Klasse wird in C# immer durch das Schlüsselwort[8] `class`, gefolgt vom Namen dieser Klasse, definiert. Der »Inhalt«, d.h. die Methoden und Daten einer Klasse, werden anschließend – von geschweiften Klammern umrahmt – angefügt.

```
01: class HelloWorld
02: {
03:     /*
04:        Methoden & Daten der Klasse HelloWorld
05:     */
06: }
```

Listing 2.7 Beispiel für eine Klassendefinition in C#

Im Listing 2.7 sehen Sie ein Beispiel für die Definition einer Klasse. Für den Klassennamen gelten bestimmte Voraussetzungen in Bezug auf die verwendeten Zeichen. Für Klassennamen sind dieselben Regeln wie für die Benennung von Bezeichnern anzuwenden. Sie finden diese in Kapitel 3.

Wichtig zu wissen ist zunächst, dass der Name der Klasse mit einem Buchstaben oder einem Unterstrich »_« beginnen muss. Im weiteren Verlauf (d.h. erst ab dem zweiten Zeichen) dürfen auch Zahlen im Namen auftauchen. Zu der Liste der ungültigen Klassennamen zählen auch Schlüsselwörter. Ist der Name eine Kombination aus Schlüsselwörtern oder aus Buchstaben und Schlüsselwörtern, ist er gültig, aber ungeschickt gewählt. Klassennamen sollten stets so gewählt werden, dass keine Verwechslung mit Sprachelementen möglich ist. Eine Auflistung aller Schlüsselwörter finden Sie ebenfalls im Kapitel 3.

2.5.2 Der Einsprungspunkt

Aber dies allein erzeugt noch keinen Einsprungspunkt, an dem die Ausführung des Programms beginnt. Um solch eine Stelle in einem Programm zu erzeugen, muss eine **Funktion**[9] hinzugefügt werden, die den Namen `Main` trägt. Statt von Funktion spricht man auch von **Methode**.

Dieser Methode müssen noch zusätzlich die Schlüsselwörter `public`, `static` und `void` vorangestellt werden. Zu beachten ist hier nicht nur die Groß- und Kleinschreibung, sondern auch die Reihenfolge!

- `public` wird im Kapitel 7 erläutert, da es zur Objektorientierten Programmierung gehört.

8 Schlüsselwörter bezeichnen Zeichenketten, die eine feste Bedeutung in einer Programmiersprache haben. Keine Schlüsselwörter sind z. B. Namen von Methoden oder Variablen.

9 Eine Funktion beschreibt eine Ansammlung von Programmcode unter einem Namen. Eine Funktion kann dabei – wie ihr mathematisches Pendant – von Parametern abhängig sein. Funktionen können auch einen Wert berechnen und diesen zurückgeben.

- `static` legt fest, dass die Methode auch ohne eine Instanz der `HelloWorld`-Klasse aufgerufen werden kann.
- `void` letztendlich bestimmt, dass die Methode keinen Wert[10] zurückgibt.

Nach dem Methodennamen folgen, von runden Klammern eingeschlossen, die Parameter der Methode. Die hier vorgestellte Implementierung der `Main()`-Methode besitzt keine Parameter. Argumente, die dem Programm beim Start auf der Kommandozeile übergeben werden können, werden – falls dies gewünscht wird – in einem Parameter der `Main()`-Methode mitgegeben. Zum jetzigen Zeitpunkt würde dies aber zu weit führen. Ich führe dieses Thema daher ebenfalls in einem späteren Kapitel weiter aus.

Nach der Parameterliste folgt der **Rumpf** der Methode; dieser ist wiederum von geschweiften Klammern eingeschlossen. Er beinhaltet den Code, der ausgeführt wird, wenn man das Programm startet. Im Fall von Listing 2.1 ist dies die Methode zur Ausgabe von Zeichen auf dem Bildschirm.

Ein C#-Programm kann auch mehrere Einsprungspunkte (d.h. mehrere `Main()`-Methoden) besitzen. Diese müssen jedoch in unterschiedlichen Klassen implementiert werden. Beim Übersetzen muss dann dem Compiler mitgeteilt werden, welcher Einsprungspunkt benutzt werden soll. Dies erfolgt mit Hilfe der Kommandozeilenoption `/main:<Klassenname>`. `<Klassenname>` steht dabei für den Namen der Klasse, deren Implementierung von der `Main()`-Methode genutzt werden soll. Alle weiteren werden dadurch ignoriert. Vergisst man diesen Schalter, erfolgt eine entsprechende Fehlermeldung des Übersetzers.

2.6 Zusammenhang zwischen Klassen- & Datei-Name

Im Gegensatz zur Programmiersprache Java gibt es in C# keinen Zusammenhang zwischen Klassen- und Dateiname[11]; d.h., Sie können die Klasse `HelloWorld` auch in einer Datei namens **HalloWelt.cs** speichern, übersetzen und ausführen.

Sie sollten aber darauf achten, Ihren Dateien sprechende Namen zu geben, die nach Möglichkeit keine Sonderzeichen oder Leerzeichen enthalten. Sonderzeichen können insbesondere im Zusammenhang mit Systemen zur Versionsverwaltung oder zur automatischen Generierung von neuen Versionen zu Problemen führen, die zu beheben unnötigen Aufwand erfordern würde.

10 Gibt die Methode `Main()` einen Wert zurück, so ist dies automatisch der Rückgabewert des Programms (z. B. ein Fehlerwert). Der Rückgabewert einer Methode wird in einem späteren Kapitel behandelt. Dieser kann von Batch-Dateien auf der Kommandozeile aufgefangen und entsprechend abgearbeitet werden (Stichwort ERRORLEVEL für Batch-Programmierer). Der Rückgabewert muss vom Typ `int` (Kapitel 3) sein.

11 In Java muss die Datei genauso wie die in der Datei enthaltene Klasse benannt werden.

2.7 Ausgaben auf dem Bildschirm

```
01: public static void Main()
02: {
03:     System.Console.WriteLine("Hello, World!");
04: }
```

Listing 2.8 Die Ausgabe der Zeichenkette »Hello, World!« am Bildschirm übernimmt die WriteLine()-Methode

Im Listing 2.8 sehen Sie die Methode, die sich für die Ausgabe von Zeichen auf dem Bildschirm verantwortlich zeigt. Sie trägt den Namen `WriteLine()` und bekommt – vorerst – einen Parameter: den Text, der ausgegeben werden soll.

`WriteLine()` gibt dabei nicht nur den übergebenen Text aus, sondern setzt automatisch die Schreibmarke an den Anfang der nächsten Zeile (= Zeilenumbruch).

Der (erste) Parameter ist die auszugebende Zeichenkette; Zeichenketten – auch als Strings, **Konstanten** oder **Literale** bezeichnet – werden von Anführungszeichen eingeschlossen. Zu beachten ist dabei, dass eine Reihe spezieller Zeichen nicht in Zeichenketten vorkommen darf, da ihnen eine Spezialbedeutung zukommt. So ist das Anführungszeichen selbst oder aber auch der einfache Backslash tabu. Wie Sie diese Zeichen ausgeben können, werde ich Ihnen im Kapitel 3 zeigen.

2.7.1 Namensräume

```
01: System.Console.WriteLine(...);
```

Listing 2.9 Ein Aufruf der WriteLine()-Methode wird mit namespace und Klassennamen dekoriert

Der Aufruf von `WriteLine(...)` steht nicht »allein«, sondern er ist zusätzlich von weiteren Bezeichnern umgeben. Diese spezifizieren zunächst einen Namensraum[12], in dem die Methode (genauer gesagt, die Klasse, die diese Methode implementiert) zu finden ist. Namensräume werden verwendet, um Konflikten zwischen Klassen mit dem gleichen Namen vorzubeugen und um Programme besser logisch zu strukturieren. Wie bereits erwähnt, ist noch die Klasse, in der die Methode `WriteLine()` implementiert ist, angegeben. Namensraum, Klassenname und Methodenname werden jeweils durch einen Punkt getrennt. Die Reihenfolge ist dabei wie angegeben von links nach rechts.

Auf diese Weise wird der komplette Pfad zur Implementierung festgelegt; eine Verwechslung ist nun nicht mehr möglich.

12 Namensraum wird im Englischen *namespace* genannt.

Übertragen bedeutet dies, dass die (statische) Methode `WriteLine()` in der Klasse `Console` implementiert ist, die im Namensraum `System` abgelegt ist.

2.7.2 Umbruch der Ausgabe von WriteLine()

Ist die auszugebende Zeile länger als die Bildschirm-Zeile, wird der Text automatisch umgebrochen. Man spricht dabei von einem *harten* Umbruch, da er nicht wortweise vorgenommen wird (d.h., wie von einer Textverarbeitung her gewohnt, in einer Spalte, in der das nächste Leerzeichen steht).

`WriteLine()` ist kein Bestandteil der Sprache C#! Die Methode (eigentlich die Klasse `Console`) ist Teil der .NET-Klassenbibliothek und kann von allen Programmiersprachen, die .NET unterstützen, verwendet[13] werden!

2.8 Assembly

Früher – d.h. vor .NET – bekam man nach einem erfolgreichen Übersetzungsvorgang als Ergebnis z.B. eine ausführbare Datei (in Form eines EXE-Files). Auch andere »Formate« wie DLLs[14] oder statische[15] Bibliotheken wurden in eine Datei geschrieben.

Mit .NET führt Microsoft nun einen weiteren Abstraktionsschritt ein: Dieser wird **Assembly** (eingedeutscht als Assemblierung oder Paket) genannt. Eine Assembly ist eine Organisationseinheit, in die einzelne Komponenten, aber auch ausführbare Programme eingepackt werden können. Außer den reinen Informationen, welche Komponenten in einer Assembly enthalten sind, enthält diese noch zusätzliche Informationen – z.B. Versionsnummern. Dabei werden nicht nur die eigenen Versionsnummern gesichert, sondern auch die der zur Erstellung verwendeten weiteren Assemblies, um Abhängigkeiten auflösen zu können. Hierdurch wird es z.B. ermöglicht, dass immer die richtigen Versionen von dynamischen Bibliotheken zur Laufzeit geladen werden.

13 Man sagt sehr oft anstatt »eine Methode verwenden« auch »eine Methode rufen« (für Interessierte: In Assembler wird ein Methodenaufruf mit dem Befehl `call` eingeleitet).

14 DLL (engl. Dynamic Link Library) bezeichnet eine Möglichkeit, Code zur Laufzeit dynamisch nachzuladen und wieder zu entladen. Programme, die DLLs nutzen, sind erfahrungsgemäß sehr klein, da der Code auf weitere Dateien verteilt werden kann. Eine passende deutsche Bezeichnung existiert nicht; die Bezeichnung »DLL« hat sich auch im deutschen Sprachraum durchgesetzt.

15 Statische Bibliotheken werden zur Übersetzungszeit zum Programm hinzugebunden. Sie vergrößern die ausführbare Datei des Programms, müssen aber nicht nachgeladen werden.

Diese Informationen werden in einem **Manifest** gesichert, das mit Hilfe eines Tools aus dem .NET Framework SDK ausgelesen werden kann. Aber auch die **CLR**[16] – also die virtuelle Maschine, die das .NET-Programm überwacht, greift auf diese Informationen zurück.

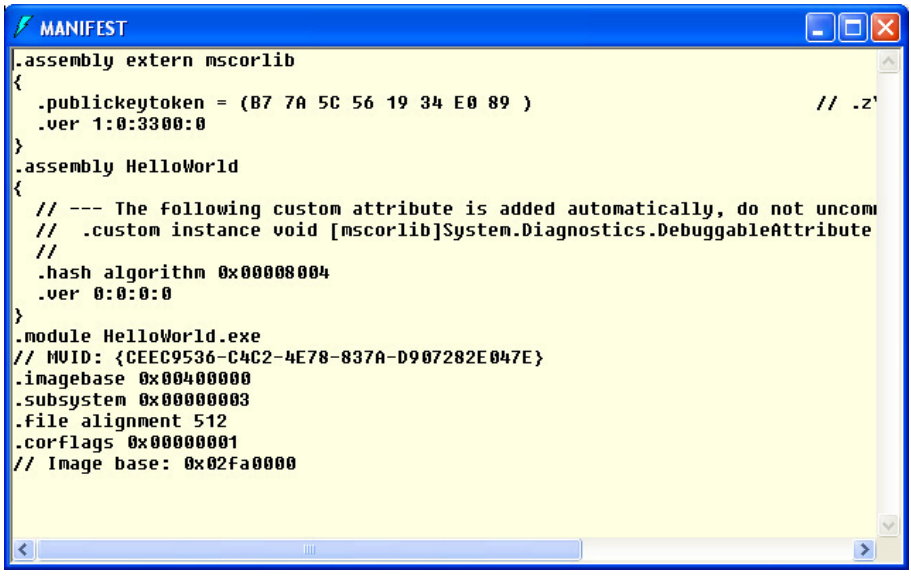


Abbildung 2.2 Das Manifest der HelloWorld-Applikation

In Abbildung 2.2 sehen Sie das Manifest der Hello-World-Applikation, wie es von **ildasm**, einem Tool aus dem .NET-Framework SDK, angezeigt wird. Deutlich sind zwei Assemblies zu erkennen:

- **mscorlib**
  Das Programm benötigt[17] die Datei *MSCORLIB.DLL*, in der die meisten Klassen des .NET-Frameworks enthalten sind – u. a. auch die Klasse `Console` mit der `WriteLine()`-Methode! Man erkennt unter dem Eintrag ».ver« deutlich die für die Erstellung dieser Applikation verwendete Version.

  Mit Hilfe dieser Versionsangabe wird .NET nun immer die richtige Version bei Ausführung der Hello-World-Applikation laden.
- **HelloWorld**
  Bezeichnet die Assembly, die durch die HelloWorld-Applikation gebildet wird. Da keine Informationen über die Versionsnummer(n)[18] hinterlegt wurden, ist

16 Common Language Runtime: Führt .NET-Programme aus.
17 Ersichtlich an dem Schlüsselwort `extern`.
18 Eine Versionsnummer setzt sich zumeist aus mehreren Teilen zusammen. Diese können getrennt voneinander angegeben werden.

auch dementsprechend nichts in die Felder für diese Informationen eingetragen bzw. die Felder wurden auf »0« zurückgesetzt.

Zu beachten ist, dass eine Assembly durchaus auch mehrere Dateien umfassen kann (**multi module assemblies**). Weiterhin unterscheidet man **private** und **shared** Assemblies.

Eine Assembly wird als »private« bezeichnet, wenn sie sich nur auf eine Applikation bezieht – normalerweise liegen dann die der Assembly angehörenden Dateien auch im selben Verzeichnis wie das das Programm selbst. Typische Beispiele für private Assemblies sind Bibliotheken, die vom Programm benötigte interne Komponenten enthalten oder für das Programm wichtige, ausgelagerte Funktionalitäten. Nutzt ein Programm nur private Assemblies, beschränkt sich das Installieren dieser Applikation auf ein einfaches Kopieren der benötigten Dateien – eine Vorgehensweise, die bereits im Zeitalter von MS-DOS gebräuchlich war.

Kann eine Assembly noch von großem Nutzen für weitere Programme sein, so empfiehlt es sich, diese Assembly allen Programmen zugänglich zu machen. Hierzu dient der so genannte **Global Assembly Cache** (= »GAC«).

Im GAC werden alle shared Assemblies nach ihren Versionsnummern und **strong names**[19] verwaltet. Hierdurch wird es möglich, mehrere Versionen derselben Bibliothek parallel auf einem Rechner zu halten – mit herkömmlichen DLLs war dies nicht möglich! Ein Programm kann dann eine bestimmte Version einer Assembly gezielt anfordern. Darüber hinaus wird eine Registrierung in der Windows-Datenbank (»Registry«) hierdurch umgangen: eine große Fehlerquelle für Nicht-.NET-Programme (fehlende Einträge oder fehlerhafte Verweise, falsche Versionen, ...).

Zum Registrieren einer Assembly im GAC gibt es im .NET-Framework ein spezielles Tool – `gacutil.exe`.

Beispiele für shared Assemblies sind z. B. die Biliotheken, die das .NET-Framework bei seiner Installation mitbringt – oder spezielle Oberflächenelemente (z. B. ein grafischer Drehregler für Audio-Applikationen). Copy-Deployment – also die Installation per Kopier-Befehl – ist bei shared Assemblies nur eingeschränkt möglich!

2.9 Zusammenfassung

Ich möchte zum Abschluss kurz zusammenfassen, was Sie in diesem Kapitel gelesen haben.

19 *strong name* (engl. starker Name): Stellt eine Möglichkeit dar, eine Komponente anhand ihres Namens eindeutig zu identifizieren. Um dies zu erreichen, werden zusätzliche Informationen zum eigentlichen Namen hinzugeneriert (z. B. ein eindeutiger Schlüssel).

2.9.1 Bestandteile eines C#-Programms

- Klasse(n):
  C#-Programme müssen mindestens eine Klasse enthalten. Dies ergibt sich aus der Ausrichtung der Sprache selbst: C# ist objektorientiert. Wie man Klassen sinnvoll einsetzt, lernen Sie im zweiten Teil dieses Buches.
- `Main()`-Methode:
  Die `Main()`-Methode ist der Einsprungspunkt des Programms; d.h., an dieser Stelle beginnt die Ausführung. Die Methode muss dabei als `public static void Main()` definiert werden. Name sowie die »Attribute« `public` und `static` müssen vorhanden sein. Zunächst erhält die `Main()`-Methode keine Parameter und gibt auch keinen Wert zurück (`void`).

2.9.2 Zusätzliches über den .NET-Framework

- `System.Console.WriteLine(...)`
  Die Klasse `Console`, die im .NET-Framework im Namensraum `System` enthalten ist, dient der Ausgabe von Zeichen auf dem Bildschirm. Die auszugebenden Zeichen werden der Methode als Parameter in Anführungszeichen übergeben. Ein Umbruch erfolgt, sobald die aktuelle Zeile kein weiteres Zeichen mehr aufnehmen kann.
- Namensräume
  Namensräume dienen der Vermeidung von Namenskonflikten und der logischen Strukturierung der Klasen und Typen eines Programms. Sie ermöglichen eine zusätzliche Qualifizierung der Klassen nach deren übergeordnetem Namensraum oder Namensräumen. Beispiele für Namensräume: `System`, `System.IO`
- Assemblies
  Assemblies sind die übergeordnete Organisationseinheit für .NET-Programme oder Bibliotheken. Mit Hilfe von Assemblies können z.B. unterschiedliche Versionen ein und derselben Datei auf einem Rechner parallel gehalten werden.

  Assemblies enthalten ein Manifest, in dem alle Parametrierungen (z.B. die Versionsnummer) festgehalten sind. Eine Assembly kann mehrere Dateien umfassen!

  Man unterscheidet zwischen private und shared Assemblies: Erstere bezeichnen Assemblies, die nur von der Applikation genutzt werden, die sie installiert hat. Letztere die Assemblies, die von allen genutzt werden können sollen. Shared Assemblies werden im Global Assembly Cache registriert, private hingegen nicht.

2.9.3 C#-Sprachelemente

- Kommentare
  Jedes Programm sollte beschreibende Kommentare enthalten. Diese erläutern schwer verständliche Programmzeilen (z. B. besonders trickreiche Algorithmen) oder geben Auskunft über die Historie einer Quellcode-Datei. Kommentare sollten knapp, aber ausreichend formuliert sein.

2.10 Übungen

Übung 2.1 *
Schreiben Sie das HelloWorld-Programm ab, übersetzen Sie es und führen Sie es aus. Funktioniert alles?

Übung 2.2 *
Prüfen Sie, welche Fehlermeldungen Ihnen der C#-Compiler meldet, wenn Sie gezielt Fehler einbauen, indem Sie das Programm aus Übung 2.1 verändern (z. B. Zeichen löschen oder zusätzliche hinzufügen). Lokalisieren Sie die Fehlerquellen anhand der erzeugten Meldungen – verstehen Sie diese! Was passiert, wenn der Text länger als die Ausgabe-Zeile ist?

Schreiben Sie das Programm aus Übung 2.1 neu und verwenden Sie für die Klasse einen anderen Namen.

Übung 2.3 *
Modifizieren Sie das Programm aus Übung 2.1 so, dass es anstatt »Hello, World!«, »Hallo« inklusive Ihrem Namen ausgibt. In meinem Fall also »Hallo, Bernhard!«.

Übung 2.4 **
Lassen Sie zusätzlich zu »Hallo, <IhrName>!« noch ein »Wie geht es Dir heute?« o. Ä. in einer neuen Text-Zeile ausgeben. Verwenden Sie hierzu einen zweiten Aufruf der `WriteLine()`-Methode!

Übung 2.5 ***
Mit Hilfe der `using`-Klausel kann man gezielt einen Namensraum »öffnen«; d. h., die enthaltenen Klassen müssen nicht mehr den vollen Pfad `<Namensraum>.<Klassenname>` tragen (`<Namensraum>` kann dabei auch eine tiefere Schachtelung, z. B. `System.IO`, beinhalten). Die Syntax von `using` ist:

```
using <Namensraum>;
```

Schreiben Sie das Programm aus Übung 2.1 (oder 2.4) so um, dass es mit Hilfe der `using`-Klausel den Namensraum *System* öffnet, und passen Sie die Aufrufe von `WriteLine()` entsprechend der neuen Situation an.

3 Konstanten, Variablen & Datentypen

Eine einfache Routine zur Text-Ausgabe macht noch lange kein fertiges Programm. Irgendwann müssen Werte eingelesen, manipuliert und zwischengespeichert werden.

Bislang kennen Sie nur einen Weg, um Zeichen auf dem Bildschirm – genauer gesagt, in einem Fenster – auszugeben. Die Ausgabe ist sicherlich ein Bestandteil eines Programms, jedoch nicht der eigentliche Kern.

Ich möchte Ihnen zu Beginn dieses Kapitels zunächst ein grundlegendes Konzept der elektronischen Datenverarbeitung vorstellen: Das **EVA-Prinzip.**

3.1 Das EVA-Prinzip

Das **Eingabe-Verarbeitung-Ausgabe-Prinzip** (oder kurz EVA-Prinzip) bezeichnet den Fluss von Daten in der elektronischen Datenverarbeitung. Zunächst müssen Daten (welcher Art auch immer) eingelesen werden, bevor sie von einer Programmeinheit verarbeitet werden können (Durchführung von Berechnungen, Erstellung von Relationen, Suchen in einer Datenbank, ...). Das Ergebnis dieser Verarbeitung wird anschließend der Ausgabeeinheit zugeführt und beispielsweise auf einem Bildschirm ausgegeben.

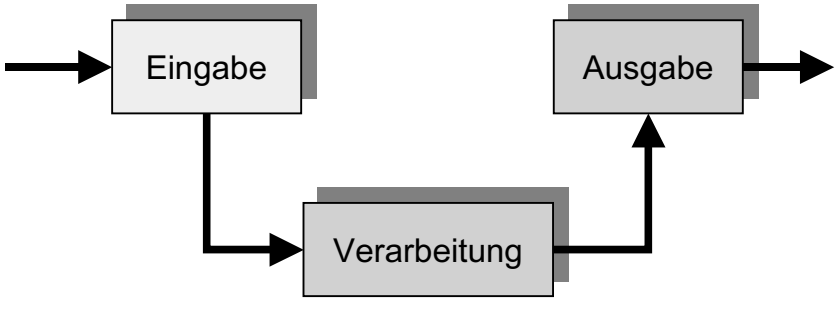


Abbildung 3.1 Das EVA-Prinzip beschreibt den Datenfluss in der elektronischen Datenverarbeitung (EDV).

In Abbildung 3.1 sehen Sie eine grafische Darstellung dieses Prinzips. Die Pfeile stellen dabei den Fluss der Daten durch das Programm dar. Abgeschaut hat man sich dies — wie im Übrigen fast alle größeren Fortschritte in der modernen Tech-

nik – von der Natur: Wir Menschen werden täglich von einer Masse an Reizen überflutet, die wir mit Hilfe unserer Sinnesorgane aufnehmen. Das Gehirn filtert diese Reize, stellt Relationen her; kurz: Es verarbeitet Daten. Die Kontraktion unserer Muskeln zur Bewegung der Gliedmaßen kann dann als eine Art »Ausgabe« aufgefasst werden!

Ob Sie sich eine Elektronik zur Steuerung eines Garagentorantriebs ansehen oder eine komplett automatisierte Fertigungsstraße in der Automobilindustrie. Es spielt keine Rolle: Überall werden Sie auf dieses für die Datenverarbeitung elementare Prinzip stoßen – auch in Computerprogrammen.

Es liefert aber – und das ist der Clou hinter dem Ganzen – äußerst hilfreiche Techniken, mit denen man Software-Projekte vereinfachen, schneller durchführen und damit auch kostengünstiger machen kann:

- Die Aufsplittung eines Problems in mehrere Teile vereinfacht die Problemlösung.
- Die Teilprobleme können getrennt voneinander (parallel) implementiert und auch getestet werden.
- Das Endprodukt setzt sich dann aus verschiedenen Einzelstücken[1], die über Schnittstellen miteinander kommunizieren, zusammen.

Das kann jedoch nur dann funktionieren, wenn die zur Kommunikation erforderlichen Schnittstellen strikt eingehalten werden. Letzteres entpuppt sich in der Praxis als sehr schwierig, da Schnittstellen sich meist während der Implementierung ändern. Ein Grund für die Änderung einer Schnittstelle kann sein, dass sich eine Spezifikation als ungünstig für die Implementierung herausgestellt hat.

Doch von der grauen Theorie wieder zurück zu C#: Im Folgenden werde ich Ihnen Mittel und Methoden vorstellen, wie der Kasten **Verarbeitung** aus Abbildung 3.1 mit C#-Mitteln gestaltet werden kann. Auch die nächsten Kapitel werden sich zumeist um die Verarbeitung von Daten drehen.

3.2 Variablen

Um Daten speichern zu können, wird Speicher benötigt. Variablen dienen dem Zugriff auf den Speicher. Am besten stellt man sich Variablen als eine Art Platzhalter für Werte während der Programmausführung vor. Auf den in einer Variablen gespeicherten Wert wird dann über einen Namen zugegriffen.

Variablen sind aber keine Möglichkeit, Daten persistent zu sichern – also in einer Form, dass diese Daten bei einem weiteren Programmstart zur Verfügung stehen.

1 Das Vorgehen, Programme aus Einzelteilen (oder auch Komponenten) zusammenzusetzen, wird in der objektorientierten Programmierung oftmals mit dem Begriff eines »Baukastensystems« verglichen.

Variablen können ihren Wert während der Ausführung eines Programms ändern. Legt man eine Variable an, wird dies im Fachjargon mit dem Begriff **Deklaration** bezeichnet.

Ich bitte Sie daher – wenn im weiteren Verlauf immer wieder die Rede von »speichern« sein wird –, dies nicht mit einer dauerhaften Speicherung auf einer Festplatte zu verwechseln.

Da in C# Variablen Objekte sind, muss bei der Deklaration einer Variablen dem Übersetzer mitgeteilt werden, was (genauer: welchen Typ Daten) sie aufnehmen soll. Gültige Typen sind zunächst die primitiven Datentypen der Sprache C#, aber auch selbst definierte Typen sind zulässig, z. B. zur Aufnahme einer Post-Anschrift oder eines Kalenderdatums. Wie Sie Datentypen selbst definieren können (sog. benutzerdefinierte Datentypen), erfahren Sie in Kapitel 4 und im zweiten Teil dieses Buches.

3.2.1 Primitive Datentypen in C#

Welche Datentypen haben Sie zunächst zur Verfügung? Im Sprachstandard für C# sind Typen für die Speicherung von Zahlenwerten (ganzzahlige und gebrochene) und logischen Zuständen (wahr oder falsch) festgelegt. Darüber hinaus gibt es einen Typ, der Zeichenketten oder allgemeiner, Text, aufnehmen kann. Im Folgenden möchte ich auf diese Typen näher eingehen.

Datentypen, die der Sprachstandard bereitstellt – d. h. Typen, die immer in einer Programmiersprache existieren –, bezeichnet man als **primitive** oder **elementare** Datentypen.

Typen, die eine Zahl aufnehmen können, unterscheiden sich hauptsächlich im möglichen Bereich, in dem eine Zahl liegen kann (Wertebereich), und in der Genauigkeit (bei gebrochenen Zahlen). Denn anders als der Mensch auf dem Papier hat der PC von seiner Konstruktion her nicht die Möglichkeit, Zahlen beliebiger Größe darzustellen. Auch die Genauigkeit bei gebrochenen Zahlen lässt manchmal zu wünschen übrig (mehr dazu im Laufe dieses Kapitels).

Neben dem Wertebereich unterscheidet man diese Datentypen auch noch in **vorzeichenbehaftete- und vorzeichenlose Datentypen**.

Vorzeichenbehaftete Datentypen können auch negative Zahlen aufnehmen, vorzeichenlose nicht!

Versucht man, eine negative Zahl einer Variablen zuzuweisen, die nur positive Werte aufnehmen kann, so erfolgt beim Übersetzen eine Fehlermeldung.

Bemerkenswert unter .NET ist weiterhin die Tatsache, dass die Datentypen Klassen sind und über zusätzliche Methoden verfügen. Diese können genutzt werden, um Informationen über einen Typ zu ermitteln (z. B. minimaler und maximaler Wert, Darstellung als Zeichenkette, …).

Der Grund dafür liegt im Prinzip der Objektorientierten Programmierung: Alle Typen sind Objekte und verfügen über einen gemeinsamen **Vater**, der bestimmte Eigenschaften an seine **Kinder** vererbt. Mehr zu diesem Thema erfahren Sie aber im zweiten Teil dieses Buches.

Alle elementaren Datentypen von C# sind auch elementare Datentypen der Intermediate Language (IL). Einziger Unterschied: In der IL tragen sie andere Namen.

3.2.2 Datentypen für ganzzahlige Werte (Menge der ganzen Zahlen)

Zunächst möchte ich Ihnen die Datentypen zur Darstellung ganzer und natürlicher Zahlen vorstellen. In Tabelle 3.1 sehen Sie eine Übersicht über diese Typen. Es sind jeweils der Name des Typs sowie minimale und maximale Werte angegeben. Es ist weiterhin aus der Tabelle ersichtlich, welche Datentypen auch negative Werte aufnehmen können und welche nicht.

Name	Minimum	Maximum
Vorzeichenlose Datentypen		
`byte`	0	255
`ushort`	0	65535
`uint`	0	4294967295
`ulong`	0	18446744073709551615
`char`	0	65535
Vorzeichenbehaftete Datentypen		
`sbyte`	-128	127
`short`	-32768	32767
`int`	-2147483648	2147483647
`long`	-9223372036854775808	9223372036854775807

Tabelle 3.1 Datentypen für die Darstellung ganzer und natürlicher Zahlen in C#

char

Der Datentyp `char` nimmt eine Sonderstellung ein: Er ist eigentlich nicht für die Aufnahme von Zahlen gedacht, sondern für alphanumerische Zeichen (Buchstaben, Ziffern, ...). Früher wurde durch den Datentyp `char` der erweiterte ASCII-Zeichensatz mit seinen 256 Symbolen abgedeckt. Mit zunehmender Internationalisierung wurden größerer Zeichensätze erforderlich, die auch andere Schriftzeichen[2] enthalten. Dieser neue Zeichensatz heißt **Unicode** und umfasst 65536 Zeichen. Daraufhin hat man diesen Datentyp um zusätzliche 8 Bit auf insgesamt 16 Bit erweitert.

Die ursprüngliche Version des ASCII-Zeichensatzes mit 7 Bit findet sich auch im neuen, großen Unicode-Zeichenpool wieder: Auch die Zeichencodes selbst stimmen immer noch überein!

3.2.3 Datentypen für gebrochene Werte (Menge der rationalen Zahlen)

Nachdem Sie nun die Typen für die Darstellung ganzer Zahlen kennen gelernt haben, möchte ich Ihnen nun die für gebrochene Zahlen (rationale Zahlen) erläutern. Hierzu möchte ich Sie zunächst kurz in die Problematik der internen Darstellung einer gebrochenen Zahl einführen.

Über die interne Darstellung von Gleitkommazahlen

Sicher haben Sie bereits viel über Bits und Bytes gehört. Was aber den meisten eher unbekannt sein dürfte, ist die Darstellungsart von Zahlwerten im Computer. Der PC verwendet seit seiner Geburtsstunde das Binärsystem, um Zahlen darzustellen und um Berechnungen auszuführen. Ich möchte Ihnen einen kurzen Einblick geben, warum man besonders im Zusammenhang mit gebrochenen Zahlen vorsichtig sein sollte. Daraus ergibt sich die Schlussfolgerung, warum man nicht allen Ergebnissen trauen sollte, die ein Computer erzeugt.

Nehmen wir als Beispiel eine einfache Berechnung: 0.5 – 0.4; spätestens seit der Einführung der Bruchrechnung in der Schule weiß man, dass das Ergebnis 0.1 ist. Führt man die Berechnung mit C# aus, erhält man 0.09999... als Ergebnis dieser Operation (Listing 3.1). Warum?

Des Rätsels Lösung liegt in der internen Darstellung: Auch gebrochene Zahlen werden durch eine Kombination der Zeichen 0 und 1 (also einer binären Zeichenkette) dargestellt. Das Problem liegt darin, dass sich bei bestimmten Brüchen – z. B. auch bei 0,1 – eine periodische Darstellung als Binärzahl auf Grund der verwendeten Methodik ergibt[3]. Da aber nicht genügend Platz für unendlich viele Bits

2 Z. B. für verschiedene asiatische Sprachen oder Hebräisch, Russisch, Griechisch, Arabisch, ...
3 0.1 dezimal = 0.0001100110011001.... binär

ist, ist die Genauigkeit entsprechend eingeschränkt, da ab einer bestimmten Stelle ganz einfach »abgeschnitten« wird. Das Resultat ist der oben erwähnte Fehler.

Geht es um Software, die für die Konstruktion einer Brücke verwendet wird, so sollte auch auf das Thema Gleitpunktarithmetik besonderes Augenmerk gerichtet werden. Meist wird dann eine eigene Zahlendarstellung programmiert.

```
01: using System;
02:
03: class CalcTest
04: {
05:   public static void Main()
06:   {
07:      float x = 0.5f;
08:      float y = 0.4f;
09:      Console.WriteLine("Ergebnis: {0}", x - y);
10:   }
11: }
```

Listing 3.1 Das Programm aus FloatingPointCalculation.cs zeigt, dass sich auch ein Computer bei der Berechnung scheinbar einfacher Werte nicht richtig verhalten muss

Der Code aus Listing 3.1 zeigt ein einfaches Programm, das den oben beschriebenen Fehler demonstriert. Es enthält einige neue Aspekte, die jedoch im weiteren Verlauf dieses Kapitels angesprochen werden.

Gleitkommadatentypen

In Tabelle 3.2 sehen Sie alle Typen aufgelistet, die für die Darstellung von Gleitkommazahlen in C# verwendet werden können.

Name	Minimum	Maximum
`float`	-3.402823E+38	3.402823E+38
`double`	-1.79769313486232E+308	1.79769313486232E+308
`decimal`	~ 1.0E-28	~ 7.9E+28

Tabelle 3.2 Die Datentypen zur Darstellung gebrochener Zahlen in C#

Gleitkommawerte werden dabei als Wert mit Exponent angegeben. Der Wert des Exponenten steht hinter dem Buchstaben `E`. Die Angabe `3.402823E+38` entspricht somit dem Zahlenwert `3.402823 * 10`^{38}.

Der Typ `decimal` nimmt eine Sonderstellung ein: Er zeichnet sich durch eine höhere Genauigkeit (max. 29 Stellen), aber im Vergleich zu `float` und `double` kleinen Wertebereich aus.

3.2.4 Wahrheitswerte

Wahrheitswerte werden oft für Verzweigungen in Programmabschnitten benötigt. In ihnen wird gespeichert, ob z. B. eine Bedingung zutrifft ("`true`") oder nicht ("`false`"). C# geht dabei anders vor als andere Programmiersprachen: Anstatt Zahlenwerte für wahr (= `true`) bzw. falsch (= `false`) festzulegen, hat man zwei Konstanten – eben `true` bzw. `false` – eingeführt.

Wahrheitswerte können über entsprechende Operatoren auch miteinander verknüpft werden. Durch eine Verknüpfung können dann komplexere Abfragen bzw. Berechnungen konstruiert werden, wie sie z. B. in der booleschen Algebra vorkommen.

Der Datentyp zur Repräsentation eines Wahrheitswertes in C# heißt `bool`. Er kann, wie bereits erwähnt, nur die Werte `true` oder `false` annehmen.

3.2.5 Zeichenketten

Will man ganze Texte sichern oder z. B. auch nur ein einzelnes Wort, benötigt man einen speziellen Datentyp: `string`[4].

Mit C# bzw. unter .NET hat man sich auf eine Darstellungsform geeinigt. Zeichenketten sind in .NET mit einer Längenangabe versehen; d. h., das Ende einer Zeichenkette wird nicht wie in C/C++ durch ein spezielles Zeichen angegeben. Diese Darstellung bietet den Vorteil, dass das Ende einer Zeichenkette beim Parsen[5] nicht anhand eines überschreibbaren Zeichens erkannt werden muss. Fehlt dieses Ende-Zeichen nämlich, können Fehler wie Bufferoverflows[6] auftreten, die u. U. Sicherheitslücken verursachen können.

4 Engl. Faden, Schnur: Einzelne Zeichen sind wie Perlen auf einer dünnen Schnur aufgefädelt.

5 Unter »Parsen« versteht man die zeichenweise Untersuchung eines Textes auf bestimmte Schlüsselbegriffe oder -zeichen. Ein HTML-Parser ist beispielsweise ein Stück Programm, das die HTML-Tags erkennt und entsprechende Aktionen auslösen kann.

6 Bufferoverflow (engl. Pufferüberlauf): Der zur Verfügung gestellte (Puffer-)Speicher reicht nicht aus. Eine Schreiboperation ersetzt nachfolgenden Daten- oder Programmcode! Meist durch entsprechende Sorgfalt beim Programmieren zu verhindern, oftmals aber einfach übersehen.

3.2.6 Namenskonventionen für Variablen

Möchte man Variablen deklarieren, muss zunächst ein Name für diese Variable gefunden werden. Dieser muss einer bestimmten Konvention genügen, da ihn sonst der Compiler nicht akzeptiert.

Regeln nach dem Sprachstandard

Den Namen einer Variablen nennt man **Bezeichner**. Gültige Bezeichner sind in C# alle Zeichenketten, die mit einem Unterstrich »_« oder mit einem Buchstaben beginnen. In ihrem weiteren Verlauf dürfen Zahlen und Buchstaben, beliebig kombiniert, auftreten[7]. Sonderzeichen wie Semikolon, Doppelpunkt oder Leerzeichen sind für Bezeichner tabu!

Nun kann man aufgrund dieser Definition viele Namen zusammenstellen. Ausnahmen bilden die so genannten **Schlüsselwörter**: Diese sind reserviert und dürfen daher nicht als Bezeichner verwendet werden.

abstract	base	bool	break	byte
case	catch	char	checked	class
cons	continue	decimal	default	delegate
do	double	else	enum	event
explicit	extern	false	finally	fixed
float	for	foreach	goto	if
implicit	in	int	interface	internal
is	lock	long	namespace	new
null	object	operator	out	override
params	private	protected	public	readonly
ref	return	sbyte	sealed	short
sizeof	static	string	struct	switch
this	throw	true	try	typeof

Tabelle 3.3 Schlüsselwörter der Sprache C# dürfen nicht als Bezeichner verwendet werden.

7 Da C# eine Sprache ist, die Unicode unterstützt, darf auch jedes Unicode-Zeichen Teil eines Bezeichners sein. Zwecks besserer Lesbarkeit sollten Sie aber auf diese Möglichkeit verzichten und nur »normale« Zeichen verwenden. Unicode-Zeichen können Sie mit Hilfe der Kombination *uxxxx* in einen Bezeichner einbauen. *xxxx* steht dabei für den Hexadezimalcode des entsprechenden Zeichens in der Unicode-Tabelle.

uint	ulong	unchecked	unsafe	ushort
using	virtual	void	while	

Tabelle 3.3 Schlüsselwörter der Sprache C# dürfen nicht als Bezeichner verwendet werden. (Forts.)

Man hat im Sprachstandard von C# die Möglichkeit vorgesehen, dass ein Quelltext, der in einer anderen Programmiersprache geschrieben ist, Bezeichner beinhalten kann, die Schlüsselwörtern aus Tabelle 3.3 entsprechen. Um großen Aufwand für die Änderung von Bezeichnern zu vermeiden, kann man durch Voranstellen des @-Zeichens ein Schlüsselwort auch als Bezeichner »missbrauchen« und diesem Konflikt so aus dem Weg gehen.

Ich beschreibe dies absichtlich mit drastischen Worten, da der geschilderte Fall die einzige Rechtfertigung für die Existenz dieser Regelung im Standard darstellt. Wenn Sie ein neues Projekt aufsetzen, sollten Sie daher immer Bezeichner verwenden, die nicht mit Schlüsselwörtern kollidieren[8].

Schlüsselwörter besitzen in einer Programmiersprache eine feste Bedeutung, die zumeist mit der Definition von Typen oder der Steuerung des Programmablaufs verbunden ist. Auch die Namen der elementaren Datentypen sind Schlüsselwörter!

Beispiele für Bezeichner

In Tabelle 3.4 sehen Sie Beispiele für gültige und ungültige C#-Bezeichner. Decken Sie beim Durchgehen der Tabelle die rechte Spalte ab und überlegen Sie sich die Antwort auf die Frage nach der Gültigkeit selbst. Ihre Ergebnisse können Sie dann mit der Tabelle vergleichen.

Bezeichner	Gültig (j/n)
Hallo	j
hallo	j
_hallo	j
_123	j

Tabelle 3.4 Beispiele für gültige und ungültige C#-Bezeichner

8 Dies kann u.U. sehr schwierig werden, da andere Sprachen wiederum anders lautende Schlüsselwörter verwenden können.

Bezeichner	Gültig (j/n)
`123Hallo`	n (beginnt mit einer Ziffer)
`checked`	n (Schlüsselwort)
`Hallo_Welt`	j
`_hallo_welt_2002`	j
`variable_1`	j
`variable 2`	n (Leerzeichen im Namen)
`variable;3`	n (Semikolon)
`@if`	j

Tabelle 3.4 Beispiele für gültige und ungültige C#-Bezeichner (Forts.)

Weitere Richtlinien

Neben den Regeln, die der C#-Standard verbindlich festlegt, gibt es in der Entwicklergemeinde zusätzliche Richtlinien, wie Variablen zu benennen sind. Diese sind nicht bindend, ihre Einhaltung dient aber der besseren Lesbarkeit – insbesondere dann, wenn mehrere Entwickler an einem Projekt schreiben.

Ein Beispiel für eine solche Richtlinie ist die Benennung von Klassenmitgliedern: Manche forderten, diese Bezeichner mit `m_` zu beginnen. Eine andere Richtlinie hatte zum Ziel, Informationen über den Typ einer Variablen in den Bezeichner zu packen (z. B. ungarische Notation): `int`-Variablen müssen in dieser Notation mit dem Zeichen `n` beginnen.

Diese Richtlinien sind in der Regel von einer Firmen- oder Projektphilosophie abhängig, weshalb ich an dieser Stelle nicht weiter auf sie eingehen möchte.

3.2.7 Deklaration von Variablen

In diesem Abschnitt möchte ich die beiden Themenblöcke Datentyp und Bezeichner zu einer Einheit zusammenfügen. Daraus ergibt sich die Deklaration einer Variablen – also die Erzeugung eines Platzhalters.

```
<Datentyp> <Bezeichner>;
```

Listing 3.2 Syntax für die Deklaration einer Variablen

In Listing 3.2 sehen Sie die Syntaxschreibweise für die Deklaration einer Variablen. Als Erstes wird der Datentyp und anschließend der Variablenname (= Bezeichner) angegeben. Die Deklaration wird mit einem Semikolon abgeschlossen. Beispiele für gültige Deklarationen sehen Sie in Listing 3.3.

```
01: int x;              // int Variable mit Namen "x"
02: double pi;          // double Variable mit Namen "pi"
03: ulong high_value;   /* ulong Variable mit Namen
04:                        "high_value" */
```

Listing 3.3 Die Deklaration von Variablen in C#

Variablen können – zunächst – innerhalb von Blöcken deklariert werden. Es ist darauf ist zu achten, dass Deklarationen nicht mit normalem Code wild gemischt werden sollten. Hier gehen die Meinungen aber auseinander. Letztendlich ist es wieder persönlicher Geschmack, welche Methodik man bevorzugt. Bedacht werden sollte aber immer, dass eventuell noch andere Leute diesen Code lesen und verstehen müssen.

Will man mehrere Variablen des gleichen Typs anlegen, hat man zwei Möglichkeiten: Entweder man schreibt alle Deklarationen der Reihe nach auf (jede in einer extra Zeile), oder man fasst diese in einer einzigen Anweisungszeile zusammen. Die Syntax hierfür sehen Sie im Listing 3.4, ein Beispiel in Listing 3.5.

```
<Datentyp> <Bezeichner 1>, ... , <Bezeichner n>;
```

Listing 3.4 Die Deklaration mehrerer Variablen desselben Typs kann auch innerhalb einer einzigen Zeile erfolgen

Wählt man letztere Methode, sind die einzelnen Deklarationen durch ein Komma voneinander zu trennen. Der Datentyp muss nur einmal, am Anfang der Zeile, angegeben werden.

```
01: long x, y, z;
```

Listing 3.5 Beispiel für die Deklaration mehrerer Variablen desselben Typs

Man sollte bei dieser Methode darauf achten, dass die Lesbarkeit für Dritte erhalten bleibt. Nur logisch zusammengehörende Variablen sollten in einer Zeile zusammengefasst werden. So sind sicherlich die Koordinaten eines Punktes im Raum eine Einheit; eine Variable für die Aufnahme des Kontostandes sollte in einer neuen Zeile angelegt werden.

3.2.8 Wertzuweisung und Initialisierung von Variablen

Nachdem man eine Variable angelegt hat, muss dieser vor ihrer Verwendung ein (konstanter) Wert zugewiesen werden. Werte werden dabei in einer bestimmten Schreibweise angegeben.

> Konstante Werte, z. B. in einer Zuweisung, werden mit dem Begriff **Literal** bezeichnet.

Oftmals trifft man neben diesem Begriff auch noch den der **Konstante** an. Bei diesem Begriff scheiden sich jedoch die Geister, da es ihn noch in einem anderen Zusammenhang gibt. Ich bevorzuge daher den Begriff des Literals und werde diesen auch weiterhin verwenden.

Literale unterscheidet man wie Datentypen in ganzzahlige, boolesche, reale (gebrochene), Zeichen- und String-Literale. Der Typ des Literals muss dem Typ der Variablen entsprechen. Versucht man, ein String-Literal einer `int`-Variablen zuzuweisen, wird dies vom Compiler mit einer Fehlermeldung quittiert.

Einer Variablen wird über den Zuweisungsoperator = ein Wert (Literal) zugeordnet.

```
<Bezeichner> = <Literal>;
```

Listing 3.6 Syntax für die Zuweisung eines Wertes an eine Variable

In Listing 3.6 sehen Sie die vollständige Syntax einer Zuweisung. Den hier dargestellten Vorgang bezeichnet man als **Wertzuweisung**.

An Stelle des Literals kann auch ein Funktionsaufruf stehen. Das Ergebnis der Funktion wird dann der Variablen zugewiesen.

Man kann eine Wertzuweisung auch als eine Methode mit Seiteneffekt auffassen. Dies bedeutet, dass die Methode (Wertzuweisung) eine Variable, die außerhalb ihres Bereichs liegt, verändert. Daraus resultiert die Tatsache, dass der Rückgabewert – also der Wert der Zuweisung – der zugewiesene Wert selbst ist. In den Übungen zu diesem Kapitel finden Sie zu dieser Thematik eine kleine Aufgabe.

Ganzzahlige Literale

Unter ganzzahligen Literalen versteht man den eigentlichen Zahlenwert; d.h., die Zahl wird, wie man es in der Schule gelernt hat, niedergeschrieben. Ein Vorzeichen muss nur bei negativen Zahlen angegeben werden.

Der Wert wird automatisch, dem Wertebereich entsprechend, in ein Datum vom Typ `int`, `uint`, `long`, ... umgewandelt.

```
123, -32, 99, 0
```

Listing 3.7 Beispiele für ganzzahlige Literale

Oftmals werden Werte im Hexadezimalsystem angegeben. Um ein ganzzahliges Literal im Hex-System zu erhalten, muss bei dieser Form die Zeichenfolge `0x` der eigentlichen Zahl vorangestellt werden.

```
0x123, -0x23, 0xAFFE
```

Listing 3.8 Hex-Zahlen können auch als ganzzahlige Literale verwendet werden

Listing 3.8 zeigt Beispiele für Literale im Hexadezimalsystem. Wie man Werte ins Hexadezimalsystem umrechnet bzw. den Wert einer Hex-Zahl im Dezimalsystem bestimmt, finden Sie in einschlägiger Informatik-Fachliteratur oder auch in jedem besseren Mathematikbuch mit einem Abschnitt über Zahlensysteme. Eine schnellere Möglichkeit ist, den Windows-Taschenrechner zu verwenden. Dieser kann – wenn er auf wissenschaftliches Rechnen umgestellt wurde – zwischen den wichtigsten Zahlensystemen konvertieren (Hex, Oktal, Binär und Dezimal).

Die Darstellung von Hex-Zahlen ist nicht case-sensitive, d.h., es erfolgt keine Unterscheidung in Groß- oder Kleinschreibung der Buchstaben A–F[9]. Auch eine Mischung aus beiden ist möglich.

Reale (Gleitkomma-)Literale

Reale Literale werden für Gleitkommazahlen verwendet. Im Sprachstandard ist der Typ von realen Literalen mit `double` festgelegt. Erst durch Anhängen von zusätzlichen Qualifizierern kann man Literale vom Typ `float` oder `decimal` erzeugen.

```
<Zahlwert>[<Exponent>][Typ]
```

Listing 3.9 Die Syntax zur Darstellung von realen Literalen ist komplexer als die von ganzzahligen Literalen

Beachten Sie bitte in Listing 3.9, dass der Typ optional anzugeben ist. Wird er ausgelassen, wird der Typ `double` angenommen. Listing 3.10 zeigt Beispiele für reale Literale.

```
1.299        // Typ = double
0.345e2      // Typ = double, Wert = 34.5
123.23f      // Typ = float
56.34e-2f    // Typ = float, Wert = 0.5634
```

Listing 3.10 Beispiele für reale Literale

Tabelle 3.5 enthält eine Übersicht über die zur Verfügung stehenden Typqualifizierer.

Typ	Spezifizierer
`double`	`d`, `D`, per default
`float`	`f`, `F`
`decimal`	`m`, `M`

Tabelle 3.5 Typqualifizierer für reale Literale in C#

9 Im Hexadezimalsystem werden neben den Ziffern 0–9 noch die Buchstaben A–F für die Darstellung von Zahlenwerten (10–15) verwendet.

Die Default-Einstellung des Typs `double` bedeutet, dass beim Programmieren reale Literale, die einer `float`-Variablen zugewiesen werden sollen, immer zusätzlich mit dem entsprechenden Qualifizierer aus Tabelle 3.5 zu versehen sind. Man sollte aber nicht auf die Nutzung von `float`-Variablen verzichten und stattdessen `double` Werte einsetzen: `float`s gehen sparsamer mit dem Speicher um.

Die Ausgabe von realen Literalen kann wie die Eingabe und wie in Listing 3.10 gezeigt auch in Exponentenschreibweise erfolgen.

Boolesche Literale

Die booleschen Literale `true` und `false` habe ich bereits in Abschnitt 3.2.4, *Wahrheitswerte,* eingeführt.

Zeichenliterale

Dem Datentyp `char` können einzelne Zeichen zugewiesen werden. Ein Zeichenliteral wird durch einfache Anführungszeichen umschlossen. So genannte Escapesequenzen stellen zusätzliche Zeichen zur Verfügung, die man nicht auf der Tastatur findet oder die Teil der Syntax sind.

Escapesequenzen bestehen aus einem Backslash[10] \ und nachfolgenden Zeichen. Sie bilden die einzige Ausnahme, in der mehrere Zeichen innerhalb der einfachen Anführungszeichen stehen dürfen.

```
'<Zeichen>' oder
'<Escape Sequenz>'
```

Listing 3.11 Syntax eines Zeichenliterals

In Tabelle 3.6 sehen Sie eine Auflistung aller Escapesequenzen sowie deren Bedeutung.

Sequenz	Bedeutung
`\'`	einfaches Anführungszeichen
`\"`	doppeltes Anführungszeichen
`\\`	Backslash
`\0`	Null-Zeichen[11]
`\a`	Alarm (kurze Tonausgabe)

Tabelle 3.6 Auflistung der Escapesequenzen

10 Der Backslash ist als umgedrehter Schrägstrich auf der Tastatur zu finden.

11 Das Nullzeichen ist ein spezielles Zeichen, das an der ersten Stelle in allen Zeichentabellen (Unicode & ASCII) zu finden ist. Es ist verschieden vom Zeichen »0« für eine Zahl!

Sequenz	Bedeutung
\b	Backspace (Rückschritt)
\f	Vorschub
\n	Neue Zeile
\r	Wagenrücklauf
\t	Tabulator
\v	vertikaler Tabulator
\xdddd	Zeichen (dddd steht für den Hex-Code des Zeichens)
\udddd	Unicode-Zeichen dddd (Hex-Code)

Tabelle 3.6 Auflistung der Escapesequenzen (Forts.)

In Listing 3.12 sehen Sie Beispiele für Zeichenliterale und die Verwendung von Escapesequenzen.

```
01: char t, x, y, z;
02: x = 'a';
03: y = '\\';          // Backslash
04: z = '\x0044';      // entspricht z = 'D'
05: t = '\'';          // einfaches Anführungszeichen
```

Listing 3.12 Beispiele für Zeichenliterale

Stringliterale

Stringliterale können als Text oder Zeichenketten aufgefasst werden. Die Zeichenkette ist dabei von doppelten Anführungszeichen einzuschließen. Will man innerhalb der Zeichenkette Sonderzeichen wie das Anführungszeichen ausgeben, muss man Escapesequenzen verwenden.

```
"<Zeichenkette mit Escapesequenzen>"
```

Listing 3.13 Syntax-Schreibweise für Stringliterale

Ein Beispiel für die Syntax aus Listing 3.13 sehen Sie in Listing 3.14.

```
"Dieser\tText\tist\tdurch\tTabulatoren\tgetrennt"
```

Listing 3.14 Beispiel für eine Zeichenkette, die Escapesequenzen enthält

Wertzuweisungen

Damit lässt sich nun endgültig die Wertzuweisung einführen. Bereits zu Beginn dieses Abschnitts haben Sie in Listing 3.6 die Syntax für eine Wertzuweisung kennen gelernt. In Listing 3.15 sehen Sie Beispiele für gültige Wertzuweisungen.

Achten Sie beim Lesen dieses Beispiels auch darauf, wie die Variablendeklarationen gruppiert wurden.

```
01: int x, y;
02: x = 123;
03: y = 0x123a;
04:
05: float f;
06: double d, e;
07: f = 1.23459f;
08: d = 0.4711e4;
09: e = 0.815;
10:
11: char c;
12: c = 'g';
13:
14: string s1, s2;
15: s1 = "Hallo, Welt!";
16: s2 = "Du hast \"Hallo\" zu mir gesagt!";
17:
18: bool b;
19: b = true;
```

Listing 3.15 Beispiele für die Wertzuweisungen

Wertzuweisungen können jederzeit innerhalb eines Programms gemacht werden. Als Ergebnis einer Wertzuweisung wird immer der zugewiesene Wert zurückgegeben.

```
01: int x = 2;
02: Console.WriteLine(x = 4);
```

Listing 3.16 Das Ergebnis einer Wertzuweisung wird ausgegeben

Listing 3.16 zeigt ein Beispiel dafür. Ich habe hier der Ausgabe von Variablenwerten, die weiter unten folgt, vorgegriffen. Im Moment reicht es zu wissen, dass Zeile 2 das Ergebnis der Zuweisung `x = 4` ausgibt.

Initialisierung einer Variablen

Bislang habe ich in den Beispielen immer zuerst eine Variable angelegt und diese in einer der nächsten Zeilen mit einem Wert belegt. Dies kann auch kürzer in einer Zeile eines Programms erfolgen.

> Unter der Initialisierung einer Variablen versteht man das Zuweisen eines Wertes bei der Deklaration.

```
<Datentyp> <Bezeichner> [= <Literal>];
```

Listing 3.17 Erweiterte Syntax einer Variablendeklaration

Listing 3.18 zeigt ein Beispiel für die Initialisierung einer Variablen. Man sieht, wie eine Variable vom Typ `int` angelegt und der Wert `4711` zugewiesen wird.

```
01: int x = 4711;
```

Listing 3.18 Initialisierung einer Variablen

Variablen sollten sich immer in einem definiertem Anfangszustand befinden. Es ist daher unumgänglich, Variablen vor ihrer Verwendung zu initialisieren.

Ähnlich wie bei der Variablendeklaration können auch hier mehrere Variablen pro Zeile initialisiert werden. Ein Beispiel dazu sehen Sie in Listing 3.19.

```
01: int x = 3, y, z = 9;
```

Listing 3.19 Auch mehrere Initialisierungen pro Zeile sind legitim

Man sieht, dass nicht alle Variablen einer Deklarationszeile mit einem Wert vorbelegt werden müssen. Deklarationen dieser Art sollten aber getrennt aufgeschrieben werden, da diese Schreibweise zu einer wesentlich schlechteren Lesbarkeit führt.

3.2.9 Ausgabe von Variableninhalten

Ich habe Ihnen in den letzten Abschnitten gezeigt, wie Sie Werte während der Programmausführung speichern können. Irgendwann will man aber auch diese Werte wieder am Bildschirm ausgegeben. Um diese Thematik dreht sich dieser Abschnitt.

Für die Ausgabe eines Variablenwertes verwendet man, wie zur Ausgabe eines Textes, die Funktion `WriteLine()`. Dieser wird zunächst ein Text als erster Parameter übergeben, der an der Stelle, an der später die Variablenwerte stehen sollen, Platzhalter enthält. Als weitere Parameter werden dieser Funktion die auszugebenden Variablen übergeben. Alle Parameter werden dabei durch Kommata voneinander getrennt angegeben.

```
WriteLine(<Format-String>[,<Var 0>[,<Var 1>[,...<Var n>]]]);
```

Listing 3.20 Diese Version der WriteLine()-Funktion erlaubt die Ausgabe von Variablenwerten

<Format-String> steht für den auszugebenden Text mit Platzhaltern für die Variablenwerte. Ein Platzhalter für eine Variable besteht aus einer von geschweiften Klammern umgebenen Nummer. Diese gibt die Position der zugehörigen Variablen in der Parameterliste an. Die Nummerierung wird, wie aus Listing 3.20 ersichtlich ist, bei 0 mit dem ersten Parameter begonnen.

```
01: using System;
02: class OutputDemo
03: {
04:    public static void Main()
05:    {
06:       int x = 4711;
07:       Console.WriteLine("Wert von x = {0}", x);
08:    }
09: }
```

Listing 3.21 Das Programm OutputDemo.cs veranschaulicht die Ausgabe von Variablenwerten mit Hilfe der Funktion WriteLine()

Ein weiterer auszugebender Parameter würde nun mit {1} bezeichnet, folgende dementsprechend weiter durchnummeriert. Ändert man daher die Zeile 7 um in

```
07: Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

und fügt zusätzlich die Variablen y und z durch die beiden Zeilen

```
06a: float y = 3.1415f;
06b: long  z = 4711081547110815;
```

in das Programm ein, so werden nun anstatt nur einer Variablen gleich drei ausgegeben. Man sieht, dass die auszugebenden Variablen nicht vom gleichen Typ sein müssen. Weiterhin erwähnenswert ist die Tatsache, dass die Platzhalterreihenfolge nicht festgelegt ist.

```
07: Console.WriteLine("{2}, {1}, {0}", x, y, z);
```

Listing 3.22 Die Reihenfolge der Platzhalter im Format-String ist nicht festgelegt und kann daher beliebig vertauscht werden

Tauscht man wiederum Zeile 7 aus Listing 3.21 gegen die aus Listing 3.22 aus, so wird zunächst der Inhalt von z, dann der von y und zuletzt der von x ausgegeben.

Sie finden das Programm OutputDemo.cs im Unterverzeichnis OutputDemo auf der CD wieder. Darin enthalten sind noch zwei weitere Programme, die die oben angesprochenen Änderungen Schritt für Schritt enthalten (OutputDemo2.cs und OutputDemo3.cs).

Sie können auch mehr als drei Parameter ausgeben. Probieren Sie einfach einmal aus, wie viele Sie ausgeben können ... Dazu sei aber gesagt, dass es nicht sinnvoll ist, so viele Werte in einer Zeile auszugeben!

Formatierte Ausgabe von Variablenwerten

Oftmals ist es erforderlich, einen Wert bei der Ausgabe mit besondere Merkmalen darzustellen: So kann z. B. festgelegt sein, dass ein Wert eine definierte Breite hat, so dass sich eine Tabellenform bei der Ausgabe ergibt.

Dazu wendet man die Syntax für die Angabe von Platzhaltern in `WriteLine()` aus Listing 3.23 an.

```
{M[,N][:<Format>]}
```

Listing 3.23 Vollständige Syntax für die Angabe eines Platzhalters in der WriteLine()-Methode

`M` bezeichnet die Nummer des Parameters. Der optionale Parameter `N` gibt die Mindestzahl der Stellen an, die für die Ausgabe verwendet werden soll. Wird eine positive Zahl übergeben, wird die Ausgabe, falls diese weniger Zeichen besitzt, rechtsbündig im Feld mit der Breite `N` ausgerichtet. Wird eine negative Zahl übergeben, erfolgt die Ausgabe trotzdem mit der (positiven) Zahl Stellen, die Ausrichtung ist aber linksbündig.

Für `<Format>` gültige Werte können Sie Tabelle 3.7 entnehmen.

Wert	Beschreibung
`C` oder `c`	Währung (currency). Formatiert eine Zahl in eine Zeichenkette, so dass diese eine Währung darstellt. Die Einheit der Währung wird automatisch an die Zeichenkette angehängt (z. B. $ für USA). Damit dies funktioniert, muss dem System mitgeteilt werden, welche Ländereinstellungen verwendet werden sollen. Hierzu zieht man die Namensräume `System.Threading und System.Globalization` mittels `using`-Klausel ein. Danach setzt man im aktuellen Thread (Kapitel 15) noch die Ländereinstellungen um. Dies erfolgt über die Anweisung `Thread.CurrentThread.CurrentCulture = new CultureInfo("de-DE");`. Alternative für `de-DE` (Deutschland) ist z. B. `en-US` für die USA. Alle `WriteLine()`-Methodenaufrufe mit dem Formatspezifizierer `C` werden dann in der entsprechenden Währung ausgegeben. Die Ausgabe des Euro-Zeichens hängt stark von den verwendeten Zeichensätzen ab. Es ist möglich, dass anstelle des Euro-Zeichens € ein anderes Zeichen ausgegeben wird.
`D` oder `d`	Ganze Zahlen (decimal). Kann nur auf ganzzahlige Datentypen angewendet werden. Die Angabe der Mindestbreite gibt die Anzahl der Stellen bei der Ausgabe an.

Tabelle 3.7 Vordefinierte Formatspezifizierer

Wert	Beschreibung
E oder e	Exponentielle Darstellung. Jede Zahl kann als Fließkommazahl in der Form k.dddExxx angegeben werden. k, d und x beschreiben Ziffern zwischen 0 und 9, k ist eine Ziffer zwischen 1 und 9. Die Angabe k.dddExxx ist zu lesen als k.ddd * 10xxx. Es wird immer mindestens eine Ziffer vor den Dezimalpunkt gesetzt, die ungleich null ist. Die Angabe der Mindestlänge bestimmt die Anzahl der ds. Fehlt diese Angabe oder ist sie kleiner als 6, werden per Definition sechs Nachkomma-Stellen ausgegeben. Die Ausgabe von xxx erfolgt immer unter Angabe eines Vorzeichens (negativ und positiv) und mindestens drei Ziffern. Die Groß-/Kleinschreibung des Formatspezifizierers bestimmt, ob die Ausgabe des Zeichens E ebenfalls groß oder klein erscheint.
F oder f	Festkommazahlen. Gibt eine Zahl in der Form ddd.dddd aus. Die Angabe einer Mindestlänge bezieht sich auf die Anzahl der Dezimalstellen.
G oder g	Wählt die beste Darstellung aus F oder E aus.
N oder n	Nummer. Stellt eine Zahl in Form einer Zeichenkette mit Tausendertrennzeichen dar. Diese Trennzeichen werden nach jeder 3er-Zifferngruppe eingefügt.
P oder p	Prozent. An die Ausgabe wird zusätzlich das Prozentzeichen % angehängt. Vor der Darstellung wird die Zahl zunächst mit 100 multipliziert.
R oder r	Stellt sicher, dass eine Rückkonvertierung der Zeichenkette in die ursprüngliche Zahl möglich ist. Für float-Werte werden zunächst sieben Stellen konvertiert, für double 15. Ist eine Rückkonvertierung möglich, wird die Ausgabe mit Hilfe von G bzw. g formatiert. Ist eine Rückkonvertierung nicht exakt möglich, erfolgt die Umwandlung in eine Zeichenkette mit neun Stellen für float- bzw. 17 Stellen für double-Werte. Eine Vorgabe für eine Mindestzahl an Stellen wird, falls vorhanden, ignoriert.
X oder x	Wandelt die übergebene, ganze Zahl in der Ausgabe in einen hexadezimalen Wert um. Die Groß-/Kleinschreibung des Formatspezifizierers bestimmt die Art der Ausgabe der Buchstaben des Hexadezimalsystems. Es wird keine zusätzliche Kennzeichnung an der Ausgabe angebracht (z. B. 0x o. Ä.).

Tabelle 3.7 Vordefinierte Formatspezifizierer (Forts.)

Damit lassen sich bereits einige Ausgaben konstruieren. Ein paar Beispiele sehen Sie in Listing 3.24.

```
01: int x = 345;
02: double y = 23.45;
03:
04: Console.WriteLine("{0:X}", x); // Ausgabe: 159
05: Console.WriteLine("{0:e}", y); // Ausgabe: 2,345000e+001
06: Console.WriteLine("{0:R}", y); // Ausgabe: 23,45
07: Console.WriteLine("{0,-4:D}DM", x); // Ausagbe: 345 DM
```

Listing 3.24 Einige Beispiel für Formatierungsangaben

Manchmal reichen diese Angaben aber nicht aus. Man kann daher eigene konstruieren, indem man für <Format> aus der Syntaxdefinition in Listing 3.23 benutzerdefinierte Platzhalter angibt. Eine Übersicht, welche Möglichkeiten Sie mit diesen haben, enthält .

Formatierungszeichen	Bedeutung
0	Stellt einen 0-Platzhalter dar. D.h., existiert an der Stelle dieses Platzhalters in der Ausgabe eine Ziffer, so wird diese ausgegeben, anderfalls wird sie mit 0 belegt.
#	Stellt einen Ziffernplatzhalter dar. Ähnelt dem 0-Platzhalter, jedoch wird nicht 0 für eine fehlende Ziffer eingesetzt, sondern ein Leerzeichen. Führende Nullen werden nicht angezeigt!
.	Das erste Vorkommen des Zeichens . legt fest, an welcher Stelle in der Ausgabe der Dezimalpunkt gesetzt wird. Weitere .-Zeichen werden ignoriert.
,	Dem Zeichen , fallen zwei Aufgaben zu: Tausendertrennzeichen: Steht das Zeichen zwischen zwei Platzhaltern und befindet sich mindestens ein Platzhalter auf der linken Seite des Dezimaltrennzeichens, so wird nach jeder 3er-Gruppe Ziffern in der Ausgabe ein Tausendertrennzeichen eingefügt. Skalierung von Zahlenwerten: Steht das Zeichen unmittelbar links neben dem Dezimaltrennzeichens ".", so wird die Zahl vor der Ausgabe durch $1000^{\text{Anzahl der ,-Zeichen}}$ geteilt.
%	Das Vorkommen des %-Zeichens in der veranlasst eine Multiplikation der Zahl mit 100. Das %-Zeichen wird dann an der Stelle in die Ausgabe eingefügt, an der es auch in der Formatierungsangabe platziert war.
E0 E+0 e0 e+0 E-0 e-0	Bewirkt die Exponentialschreibweise für die Ausgabe einer Zahl. Die Anzahl der Nullen hinter E oder e gibt die Mindestzahl der Stellen für den Exponenten an. Die Angabe von + bedeutet, dass der Exponent immer mit einem Vorzeichen dargestellt wird. e, E, e- und E- bewirken, dass nur negative Exponenten ein Vorzeichen erhalten.
\	Auch die Formatangabe kann Escape-Zeichenfolgen enthalten. Diese werden dann an den entsprechenden Stellen in der Ausgabe platziert.
'ABC' \"ABC\"	Zeichen, die in einfachen oder durch den Backslash geschützen doppelten Anführungszeichen stehen, werden eins zu eins in die Ausgabe übernommen.
Alle weiteren Zeichen außer ";"	... werden ebenfalls in die Ausgabe übernommen.

Tabelle 3.8 Übersicht über die benutzerdefinierten Formatierungszeichen

Mit Hilfe dieser Formatierungszeichen lassen sich dann auch schön formatierte Werte ausgeben (Listing 3.25).

```
01: int x = 3456;
02: double y = 12345.6;
03:
04: Console.WriteLine("{0:00000}", x); // Ausgabe: 03456
05: Console.WriteLine("{0:#####}", x); // Ausgabe:  3456
06: Console.WriteLine("{0:#### EUR", x);  // 3456 EUR
07: Console.WriteLine("{0:###,# EUR", x); // 3.456 EUR
08: Console.WriteLine("{0:###,.###}", y); //  12,345
```

Listing 3.25 Einige Beispiele für benutzerdefinierte Formatierungszeichenketten

3.2.10 Gültigkeit von Variablen

Bislang haben Sie nur Variablen angelegt und mit einem Wert versehen, der am Ende mit Hilfe der `WriteLine()`-Methode wieder ausgegeben werden kann.

Ab wann ist aber eine Variable gültig, wann endet ihre Gültigkeit wieder und was ist bei Blöcken zu beachten?

1. Eine Variable ist von der Position ihrer Deklaration an gültig.
2. Die Gültigkeit, und damit die Lebenszeit, endet bei der nächsten schließenden Blockklammer.
3. Existiert eine Variable in einem übergeordnetem Block, so kann keine Variable in einem untergeordneten Block denselben Namen tragen.
4. Variablen, die auf derselben Schachtelungsebene, aber in unterschiedlichen Blöcken deklariert wurden, können denselben Namen tragen.
5. Variablen, die in übergeordneten Blöcken deklariert wurden, sind in untergeordneten sichtbar, umgekehrt jedoch nicht.

In Listing 3.26 sehen Sie ein Beispiel für die Missachtung der ersten Regel. Zunächst wurde in Zeile 1 die Variable für eine Ausgabe verwendet, bevor sie in Zeile 2 angelegt wird. Solche Fälle weist der Compiler mit einer Fehlermeldung bei der Übersetzung ab. Ein Fehler wird auch dann ausgegeben, wenn versucht wird, auf eine Variable, die noch nicht mit einem Wert belegt wurde, lesend (Ausgabe!) zuzugreifen.

```
01: Console.WriteLine("x = {0}", x);
02: int x = 815;
```

Listing 3.26 Eine Variable muss zuerst deklariert und initialisiert werden, bevor sie verwendet werden kann

In Listing 3.27 sehen Sie eine Veranschaulichung der Regeln 2–5.

```
01: using System;
02: class Demo
03: {
04:    public static void Main()
05:    {
06:       int x = 1;
07:       {
08:          int y = 2;
09:          int x = 2; // Fehler !!
10:
11:          Console.WriteLine("x = {0}", x);
12:          Console.WriteLine("y = {0}", y);
13:
14:          x = 3;
15:       }
16:       {
17:          int y = 4;
18:          Console.WriteLine("y = {0}", y);
19:       }
20:
21:       Console.WriteLine("x = {0}", x);
22:       Console.WriteLine("y ={0}", y); // Fehler!!
23:    }
24: }
```

Listing 3.27 VarDemo2.cs veranschaulicht die Regeln für den Gültigkeits- und Sichtbarkeitsbereiche von Variablen in C#

So würde in Zeile 9 von Listing 3.27 eine Variable in einem untergeordneten Block angelegt, die denselben Namen wie eine Variable des übergeordneten Blocks tragen würde. In Zeile 14 erkennt man, dass Variablen des übergeordneten Blocks ganz normal verwendet und auch manipuliert werden können.

In den Zeilen 16–19 sehen Sie eine Anwendung der Regel 4. Zeile 22 würde auf eine Variable zugreifen, die weder in diesem Block sichtbar ist, geschweige denn noch »lebt«.

Da auch die Funktion selbst von einem Block umrahmt wird, verliert die Variable `x` am Ende, wie alle anderen vor ihr, ihre Gültigkeit.

Tabelle 3.9 veranschaulicht, wie sich die Werte der Variablen im Programm aus Listing 3.27 in den interessanten Zeilen verhalten. Ein »-« in der Tabelle bedeutet, dass die entsprechende Variable dort nicht gültig ist.

Zeile	x	y
6	1	-
8-13	1	2
14	3	2
17	3	4
21	3	-

Tabelle 3.9 Veranschaulichung der Variableninhalte aus Listing 3.27

3.3 Felder

Oftmals werden nicht nur einzelne Werte, sondern ganze Felder von Werten (z. B. für die Darstellung einer Matrix) benötigt. Es wäre zu umständlich, für jedes einzelne Element einer Matrix eine Variable anzulegen – insbesondere im Hinblick auf Erweiterbarkeit und Fehleranfälligkeit. Auch das Durchlaufen dieser Matrix wird mit einzelnen Variablen sehr schwierig, wenn diese sehr groß sind.

```
01: int a11 = 1;
02: int a12 = 2;
03: int a21 = 3;
04: int a22 = 4;
05: int det = (a11 * a22) - (a12 * a21);
06: System.Console.WriteLine("{0}\t{1}", a11, a12);
07: System.Console.WriteLine("{0}\t{1}", a21, a22);
08: System.Console.WriteLine("Determinante = {0}", det);
```

Listing 3.28 Fragment aus Matrix.cs: Berechnung der Determinante einer 2x2-Matrix mit Hilfe der Regel von Sarrus

Auch wenn Operatoren noch nicht formal eingeführt wurden, so dürfte Listing 3.28 keine weiteren Verständnisprobleme bereiten. In den Zeilen 1 bis 4 wird eine Matrix mit zwei Reihen und zwei Spalten definiert und mit Werten belegt. Die Zahlen im Bezeichner geben jeweils die Position des Wertes in der Matrix an, wobei die Zeile zuerst genannt wird.

In Zeile 5 wird mit Hilfe der Regel von Sarrus die Determinante dieser Matrix berechnet und der Variablen `det` zugewiesen. Die Zeilen 6 bis 8 ergeben zuerst die Matrix selbst und danach den Wert der Determinanten aus.

Betrachtet man dieses Beispiel quantitativ, so wurde zunächst noch sehr wenig Code geschrieben. Erweitert man aber dieses Problem auf eine 3x3-Matrix oder noch höher, so wird sehr schnell deutlich, dass einzelne Variablen für die Speicherung einer Matrix ungeeignet sind. Heutige Problemstellungen reichen dabei an Matrizen mit mehr als 1000 Zeilen und Spalten heran – eine Sisyphusarbeit, verwendet man einzelne Variablen! Insbesondere auch im Hinblick auf die Algorithmen, die zur Berechnung von Problemen dieser Größenordnung herangezogen werden. Aus diesem Grund gibt es die Felder, im Englischen mit *Array* bezeichnet.

Unter einem Feld versteht man die Aneinanderreihung von Werten desselben Typs, wobei auf einzelne Werte über einen Index zugegriffen werden kann. Felder können ein- oder mehrdimensional sein.

3.3.1 Deklaration und Initialisierung

```
<Datentyp>[] <Bezeichner>;
<Datentyp>[] <Bezeichner> = new <Datentyp>[<Elemente>];
<Datentyp>[] <Bezeichner> = {<Elementlist>};
```

Listing 3.29 Syntax für die Deklaration eines eindimensionalen Arrays in C#

In Listing 3.29 sehen Sie die Syntax für die Deklaration eines Arrays. Bitte beachten Sie, dass an dieser Stelle die eckigen und geschweiften Klammern ("[]", "{}") in dieser Definition Teil der Syntax sind, d.h. bei der Deklaration mit niedergeschrieben werden müssen.

Ein Array wird in zwei Schritten angelegt: Zunächst wird ein Array-Bezeichner deklariert. Diesem wird anschließend ein über den `new`-Operator erstelltes Array fester Größe zugewiesen. In Listing 3.30 ist dies anhand des Arrays `a` veranschaulicht (Zeile 1 und 4).

```
01: int[] a;
02: int[] b = new int[25];
03: int[] c = {1, 3, 4, 6, 9, -33};
04: a = new int[3];
```

Listing 3.30 Beispiele für die Deklaration eines eindimensionalen Arrays

Ähnlich wie »normale« Variablen können auch Arrays initialisiert werden. Zum einen kann der Variablen direkt bei ihrer Erstellung ein Array zugewiesen werden, zum anderen kann dieses Array direkt mit Werten belegt werden. Gibt man Werte an, so bestimmt die Anzahl der Werte die Größe des Arrays[12]. Ersteres sehen Sie im Listing 3.30 am Array `b`, letzteres Vorgehen am Array `c`.

12 Die Größe wird dabei beim Übersetzungsvorgang vom Compiler ermittelt.

Sprachen wie C oder C++ erzeugen Platz im Speicher für die entsprechende Anzahl Werte. Unter .NET wird dieser Speicher noch initialisiert. D.h., nach dem Anlegen eines Arrays befinden sich alle Elemente in einem definierten Ausgangszustand. Der Initialisierungswert ist dabei ganz vom Typ der Arrayelemente abhängig.

3.3.2 Zugriff und Ausgabe von Arrays: [..], foreach

Will man auf einzelne Elemente des Arrays zugreifen, so verwendet man hierfür den Array-Zugriffsoperator `[]`. Dieser hat als einzigen Parameter in den eckigen Klammern den Index des anzusprechenden Elements.

Man beginnt mit der Indizierung nicht bei 1, sondern bei 0. Dementsprechend läuft er bis zur Anzahl der Array-Elemente minus 1.

Beispiel:

Hat man ein Array mit drei Elementen mittels `int[] a = new int[3];` deklariert, so läuft der Index von `0` bis `2` (= 3 - 1).

Versucht ein Programm, einen Zugriff außerhalb dieser Grenzen auf das Array durchzuführen, kann dies der Compiler im Fall eines negativen Indizes mit einer Warnung melden. Ist der Array-Index positiv und schlägt über die Obergrenze hinaus, wird zur Laufzeit ein Fehler von .NET generiert. Listing 3.31 veranschaulicht die richtige und falsche Indizierung eines Arrays.

```
01: int[] a = new int[3];
02: a[0] = 3;  // gueltig!
03: a[-1] = 2; // Warnung vom Compiler
04: a[9] = -1; // wird nicht vom Compiler erkannt, aber vom
05:            // Laufzeitsystem
```

Listing 3.31 Richtige und falsche Indizierung eines Arrays

Als Index für ein Array kann auch eine Variable verwendet werden. So sind beispielsweise Programmschleifen möglich, die ein Array Element für Element durchlaufen. Schleifen bzw. Programmkontrollkonstrukte sind Thema des 6. Kapitels.

Benötigt man jedes Element eines Arrays, so bietet sich das `foreach`-Konstrukt anstatt einer normalen Schleife an: Es zieht unabhängig von der Anzahl der Array-Elemente jedes Element einzeln heraus und sichert es in einer Variablen. `foreach` hat aber auch einen entscheidenden Nachteil: Die so ermittelten Elemente können nicht verändert werden[13]. Der Compiler quittiert derlei Versuche mit einer Fehlermeldung während des Übersetzungsvorgangs.

13 Diese Aussage ist richtig, solange es sich nicht um Referenzen handelt. Werden Referenzen in einem Array abgelegt, so erhält man über diese direkten Zugriff auf die dahinter liegenden Objekte. Referenzen werden im zweiten Teil dieses Buches erläutert.

```
foreach (<Elementtyp> <Bezeichner> in <Arraybezeichner>)
   <Anweisung>
```

Listing 3.32 Die Syntax des foreach-Konstrukts

Listing 3.32 zeigt die `foreach` zu Grunde liegende Syntax. Im Listing 3.33 sehen Sie die Deklaration und Initialisierung von Array-Elementen (Zeilen 5 und 6), den Zugriff auf Array-Elemente (Zeilen 8 bis 10) sowie die Verwendung von `foreach` zur Ausgabe der Feldinhalte.

```
01: class ArrayDemo1
02: {
03:     public static void Main()
04:     {
05:        int[] a = new int[3];
06:        int[] b = {1, 2, 3};
07:
08:        a[0] = 3;
09:        a[1] = 2;
10:        a[2] = 1;
11:
12:        foreach (int x in a)
13:        {
14:           System.Console.WriteLine("a: {0}", x);
15:        }
16:        foreach (int x in b)
17:        {
18:           System.Console.WriteLine("b: {0}", x);
19:        }
20:     }
21: }
```

Listing 3.33 Ausgabe von Arrays mit Hilfe des foreach-Konstrukts (ArrayDemo1.cs)

3.3.3 Mehrdimensionale Arrays

In Listing 3.28 haben Sie eine Möglichkeit kennen gelernt, Matrizen mit Hilfe einzelner Variablen darzustellen. Zweidimensionale Arrays sind insbesondere für große Matrizen wesentlich geeigneter. In diesem Abschnitt möchte ich daher noch kurz auf mehrdimensionale Arrays am Beispiel von einem Array mit zwei Dimensionen eingehen. Die Methoden, die hier vorgestellt werden, lassen sich aber auch auf Felder mit mehr als zwei Dimensionen übertragen und anwenden.

```
<Datentyp>[,] <Bezeichner>;
<Datentyp>[,] <Bezeichner> = new <Datentyp>[<AzE1>,<AzE2>];
<Datentyp>[,] <Bezeichner> = {<Elementliste>};
```

Listing 3.34 Syntax für die Deklaration eines zweidimensionalen Arrays

Will man ein Feld mit mehr als einer Dimension anlegen, muss dies dem Compiler mitgeteilt werden. Dies geschieht durch das Komma in den eckigen Klammern nach dem Datentyp (Listing 3.34). In diesem Fall wurde ein Komma angegeben, also ein zweidimensionales Feld spezifiziert. Für ein dreidimensionales Feld sind dann entsprechend zwei Kommata anzugeben. `AzEx` in Listing 3.34 steht für die Anzahl der Elemente der Dimension x. Der Indexwert wird für jede Dimension wiederum bei `0` beginnend durch die Anzahl der Elemente beschränkt.

In Listing 3.35 sehen Sie ein Beispiel für die Verwendung eines zweidimensionalen Arrays für die Aufnahme einer 2x2-Matrix.

```
01: class ArrayDemo2
02: {
03:     public static void Main()
04:     {
05:         int[,] a = new int[2,2];
06:
07:         a[0,0] = 1;
08:         a[1,0] = 3;
09:         a[0,1] = 2;
10:         a[1,1] = 4;
11:
12:         int det = (a[0,0] * a[1,1]) - (a[0,1] * a[1,0]);
13:
14:         System.Console.WriteLine("{0}\t{1}",
15:                                  a[0,0], a[0,1]);
16:         System.Console.WriteLine("{0}\t{1}",
17:                                  a[1,0], a[1,1]);
18:         System.Console.WriteLine("Determinante = {0}",
19:                                  det);
20:     }
21: }
```

Listing 3.35 ArrayDemo2.cs veranschaulicht die Verwendung von zweidimensionalen Feldern zur Darstellung von Matrizen im Rechner

Aus diesem Grund muss auch die Initialisiererliste aus der Syntaxdefinition von Listing 3.34 Arrays enthalten. Diese werden angegeben, indem man die Werte

wiederum in geschweiften Klammern niederschreibt. Somit kann man die Zeilen 5 bis 10 aus `ArrayDemo2.cs` ersetzen durch Listing 3.36.

```
05:         int[,] a = { {1,2}, {3,4} };
```

Listing 3.36 Beispiel für die Initialisierung eines zweidimensionalen Arrays

Mehrdimensionale Arrays werden in einem Block im Speicher abgelegt.

3.3.4 Unregelmäßige Arrays

Unregelmäßige Arrays sind Arrays aus Arrays, d.h., die Felder der »1. Dimension« sind wiederum Arrays. Die einzelnen Arrays dürfen dabei unterschiedlich lang sein. Diese Art von Arrays bezeichnet man im Fachjargon auch als **jagged Arrays**.

```
<Datentyp>[][] = new <Datentyp>[<Anzahl>][];
```

Listing 3.37 Die Syntax für die Definition eines unregelmäßigen Arrays

Ein kleines Beispiel für die Verwendung von jagged Arrays bzw. deren Initialisierung sehen Sie in Listing 3.38.

```
01: int[][] data = new int[2][];
02: data[0] = new int[20];
03: data[1] = new int[3];
```

Listing 3.38 Jagged Arrays enthalten Elemente, die wiederum Arrays (unterschiedlicher Größe) sind

Jagged Arrays werden auf mehrere Blöcke im Speicher verteilt (jedes Array kann an einer anderen Speicherstelle liegen).

3.3.5 Speicherbereinigung

Wenn eine Feld-Variable mit Hilfe des `new`-Operators angelegt wird, wird gleichzeitig Speicher im System als belegt markiert. Irgendwann muss dieser Speicher aber wieder an das System zurückgegeben werden. Unterbleibt dies, würden die Variablen irgendwann den kompletten Speicher des Rechners nutzen und es wäre kein Platz mehr für andere Programme.

Unter .NET gibt es den so genannten **Garbage Collector**[14]. Dieser stellt mit Hilfe ausgeklügelter Algorithmen fest, wann eine Variable nicht mehr benötigt wird, und gibt den belegten Speicher wieder frei. Für den Programmierer bedeutet dies, dass er sich nicht selbst um die Freigabe des von einem Array belegten Speichers kümmern muss.

14 Garbage Collector = Müllsammler. Der Vorgang der Garbage Collection – also des Freigebens von Speicher – ist ein äußerst komplexer und heiß diskutierter Teil in .NET. Wie genau er funktioniert, findet sich in der einschlägigen Literatur zu diesem Thema.

»Normale« Variablen hingegen unterliegen nicht der Aufsicht des Garbage Collectors. Diese werden über einen anderen Mechanismus während der Programmausführung wieder freigegeben.

3.4 Parameter der Main-Funktion

Bislang haben Sie die `Main()`-Funktion als Funktion ohne Parameter kennen gelernt. Eigentlich ein Paradoxon, berücksichtigt man die Tatsache, dass bislang nur Konsolenanwendungen geschrieben wurden. Diese können aber – wie Windows-Anwendungen auch – mit Parametern versorgt werden. Sie werden vom System als Parameter an die `Main()`-Funktion weitergeleitet.

In Kapitel 3.3., *Felder*, habe ich Ihnen Arrays vorgestellt. Da nahezu beliebig viele Parameter beim Programmstart übergeben werden können, liegt es nahe, diese als Feld an die `Main()`-Funktion zu übergeben.

```
public static void Main(string[] args)
```

Listing 3.39 Erweiterte Syntax der Main-Funktion für die Übergabe von Parametern

In Listing 3.39 sehen Sie eine Möglichkeit, wie die Syntax der `Main()`-Funktion erweitert werden kann, um einem Programm Parameter zu übergeben. Neben der hier angegeben Syntaxform gibt es noch einige weitere, die aber eine untergeordnete Rolle spielen. Diese sind z. B. in der Sprachspezifikation von C# nachzulesen.

```
01: using System;
02: class MainParam
03: {
04:    public static void Main(string[] args)
05:    {
06:       int counter = 0;
07:       foreach (string arg in args)
08:       {
09:          Console.WriteLine("Argument #{0}: {1}",
10:                            counter++, arg);
11:       }
12:    }
13: }
```

Listing 3.40 MainParam.cs gibt die Kommandozeilenparameter aus

In Listing 3.40 sehen Sie ein Beispielprogramm der Buch-CD (`MainParam.cs`), das die ihm übergebenen Parameter ausgibt.

Das Array beinhaltet immer Elemente vom Typ `string`. Sollen Zahlenwerte übergeben werden, müssen diese von einem `string` in einen entsprechenden Typ umgewandelt werden. Ein Beispiel für die Konvertierung einer Zeichenkette in einen Zahlenwert finden Sie im Abschnitt 3.6. Die Parameter werden, durch Leerzeichen voneinander getrennt, beim Programmaufruf auf der Kommandozeile angegeben.

Soll ein Parameter – z. B. eine Zeichenkette – Leerzeichen enthalten, so müssen diese auf der Kommandozeile mit Anführungszeichen quotiert werden. Näheres hierzu findet sich jedoch in der Dokumentation von Windows und soll daher an dieser Stelle nicht weiter vertieft werden.

Eine mögliche Ausgabe des Programms `MainParam.cs` aus Listing 3.40 sehen Sie in Abbildung 3.2.

In Zeile 10 von Listing 3.40 sehen Sie einen Ihnen noch unbekannten Operator ++. Er erhöht den Wert einer `int`-Variablen um 1. Eine ausführliche Beschreibung dieses Operators finden Sie in Kapitel 5.

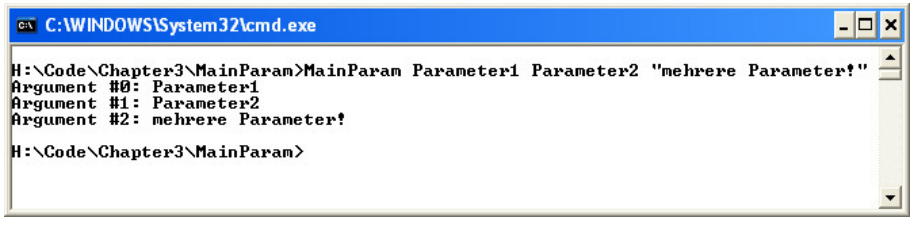


Abbildung 3.2 Programmargumente werden auf der Kommandozeile übergeben und in der Main()-Funktion ausgewertet

3.5 Typqualifizierer

Im Abschnitt 3.2.10, *Gültigkeit von Variablen*, habe ich Ihnen aufgezeigt, wie lange Variablen »leben«, d. h., bis zu welcher Stelle sie gültig sind und wann sie vom Laufzeitsystem aufgeräumt werden. **Typqualifizierer** bieten u. a. die Möglichkeit, die Lebenszeit von Variablen zu beeinflussen.

> Typqualifizierer verleihen Variablen zusätzliche Eigenschaften.

Dabei liegt allen Typqualifizierern dieselbe Syntax (Listing 3.41) zu Grunde. Sie können nur zum Zeitpunkt der Deklaration einer Variablen angegeben werden.

```
〈 Typqualifizierer〉 〈Datentyp〉 〈Bezeichner〉;
```

Listing 3.41 Erweiterte Syntax für die Variablendeklaration

3.5.1 static

`static` bezeichnet Member, die nicht mit einer Instanz verknüpft sind. Variablen, die als `static` spezifiziert wurden, verlieren erst mit Verlassen der **AppDomain**[15] ihre Gültigkeit. `static` kann nur in Verbindung mit Memberdaten eingesetzt werden. In Kapitel 7, *Einführung in die Objektorientierte Programmierung*, erfahren Sie Näheres zu diesem Thema.

3.5.2 const

Variablen, die als `const` deklariert sind, können nicht verändert werden. Einzige Ausnahme ist die Initialisierung. Man spricht in diesem Zusammenhang auch von **Konstanten**. Die Kreiszahl `pi` ist ein gutes Beispiel für eine `const`-Variable.

```
const float pi = 3.1415f;
```

Listing 3.42 Die Definition von Konstanten

Versucht man im Laufe des Programms auf diese Variable schreibend zuzugreifen, so führt dies zu einem Kompilierungsfehler (Listing 3.43).

```
const float pi = 3.1415f;
pi = 6.0f;                    // Fehler!
```

Listing 3.43 Konstanten erlauben nur lesenden Zugriff

3.6 Einlesen von Variablenwerten über die Tastatur

Sie haben bereits gehört, wie man Daten, d.h. Variablenwerte, auf dem Bildschirm ausgeben kann. In diesem Abschnitt möchte ich mich dem Einlesen von Daten über die Tastatur widmen. Bezogen auf das EVA-Prinzip aus Abschnitt 3.1 verlasse ich nun den Bereich **Verarbeitung** und gehe einen Schritt zurück zur Datenerfassung – Bereich **Eingabe** (Abbildung 3.1).

Variablenwerte – insbesondere Zahlen – können nicht direkt eingelesen werden. Um eine Zahl von der Tastatur in eine Variable zu sichern, muss man wie folgt vorgehen:

1. Einlesen des Wertes von der Tastatur in eine Variable vom Typ `string`.
2. Konvertieren des Wertes vom Typ `string` in den entsprechenden Datentyp.

Ein Beispiel hierfür sehen Sie in Listing 3.44. Zum Einlesen wird die Funktion `ReadLine()` der Klasse `Console` verwendet.

15 AppDomain = Application Domain. Organisationseinheit in .NET für Programme. Bevor die AppDomain verlassen werden kann, muss das Programm beendet worden sein. Nähere Informationen hierzu entnehmen Sie bitte der .NET-Dokumentation.

```
01: using System;
02: class ReadDataDemo
03: {
04:    public static void Main()
05:    {
06:       string input   = "";
07:       double dvalue  = 0.0;
08:
09:       Console.Write("Geben Sie eine Zahl ein: ");
10:
11:       input  = Console.ReadLine();
12:       dvalue = double.Parse(input);
13:
14:       Console.WriteLine("Sie haben {0} eingegeben.",
15:                         dvalue);
16:    }
17: }
```

Listing 3.44 Einlesen eines Wertes von der Tastatur

In den Zeilen 6 und 7 werden zunächst zwei Variablen angelegt. `input` nimmt dabei die von der Tastatur eingelesene Zeichenkette auf; `dvalue` soll dann den konvertierten Wert speichern.

In Zeile 9 wird die Funktion `Write()` zur Ausgabe einer Zeichenkette genutzt, die den Anwender auffordert, eine Zahl einzugeben. Der Unterschied zwischen `WriteLine()` und `Write()` ist, dass Erstere nach Ausgabe des Textes einen Zeilenumbruch erzeugt, Letztere nicht.

Zeile 11 und 12 beinhalten den eigentlichen Kern dieses Programms – das Einlesen und Konvertieren. In Zeile 11 wird die Funktion `ReadLine()` verwendet, um eine Zeile Text einzulesen. »Zeile« bedeutet in diesem Zusammenhang, dass so lange von der Tastatur gelesen wird, bis der Anwender die Eingabetaste drückt und dadurch einen Zeilenumbruch erzeugt. Der eingegebene Text wird anschließend in der `string`-Variablen `input` gesichert.

Dieser Wert wird dann der Klassenmethode `Parse()` der `double`-Klasse als einziger Parameter übergeben, die daraus einen Zahlenwert berechnet. Hat der Anwender eine Zeichenkette eingegeben, die keine Zahl vom Typ `double` ergibt, so wird während der Ausführung ein Fehler erzeugt und das Programm bricht an dieser Stelle ab.

In Zeile 14 wird dann der eingegebene Wert zur Kontrolle über die Funktion `WriteLine()` ausgegeben.

Dieses Vorgehen – Einlesen und anschließendes Konvertieren – ist nicht nur für Gleitkommazahlen gültig, sondern muss für jeden (anderen) Datentyp so angewendet werden (z. B. `int` oder `sbyte`).

3.7 Zusammenfassung

In diesem Kapitel haben Sie Variablen kennen gelernt. Sie wissen, wie man Variablen anlegt und auf diese zugreift. Variablen werden nach ihrem Typ unterschieden, der bei der Deklaration mit anzugeben ist. Es gibt verschiedene Typen, die sich in ihrem Wertebereich und damit auch automatisch im belegten Speicherplatz unterscheiden.

Variablen werden Literale (= Konstanten) zugewiesen. Es gibt unterschiedliche Schreibweisen für Gleitkommaliterale (`double`, `float`, `decimal`) sowie für ganzzahlige (Hexadezimalwert, Dezimalwert).

Der Typ `string` dient der Aufnahme von Zeichenketten (= Text). Zeichenketten müssen – anders als numerische Werte – durch Anführungszeichen umschlossen sein. Sie können aus normalen Zeichen und speziellen Sonderzeichen, den Escapesequenzen, bestehen.

Ich möchte Ihnen an dieser Stelle noch einige Richtlinien für die Programmierung im Umgang mit Variablen an die Hand geben, die sich im Alltag als nützlich erwiesen haben:

1. Bezeichner sollten immer den Inhalt einer Variablen beschreiben. Dadurch behalten Dritte beim Lesen Ihres Programms leichter den Überblick. Aber auch Sie werden, wenn Sie Ihren eigenen Code nach einer bestimmten Zeit wieder betrachten, leichter in das Programmgeschehen einsteigen können. Namen wie »buffi« oder »otto« helfen niemandem beim Verständnis Ihres Codes. Für Schleifenparameter (siehe Kapitel 6, *Kontrollkonstrukte*) werden hingegen sehr gerne die Bezeichner `i`, `j`, `x`, `y`, `u`, `v` oder `w` verwendet.
2. Verwenden Sie Typen, die dem Wertebereich entsprechen. Im Klartext, wenn ein Wert sich im Bereich eines `float` bewegt, dann deklarieren Sie auch eine `float`-Variable und keine vom Typ `double`.
3. `int` ist der »schnellste« ganzzahlige Datentyp. Dies liegt in der Tatsache begründet, dass die Größe eines `int` und die eines Maschinenwortes auf den vorherrschenden 32-Bit-Computern gleich sind. Ein Wert dieses Typs kann somit mit einem einzigen Lesebefehl aus dem Speicher in die CPU transferiert werden. Der hier herauszuholende Geschwindigkeitsschub ist allerdings eher marginal. Deswegen sollte man Richtlinie 2 nur verletzen, wenn triftige Gründe vorliegen! Auch muss beachtet werden, das auf künftigen 64-Bit-Systemen der

C#-Typ `int` immer noch einen 32-Bit-Wert bezeichnet, d.h., dort kann sich die Geschwindigkeitssteigerung wieder in Rauch auflösen.

4. Achten Sie auf eine klare Strukturierung der Variablendeklarationen. Gruppieren Sie zusammengehörende Werte auch optisch.
5. Initialisieren Sie mehr als eine Variable pro Zeile. Achten Sie besonders auf Einhaltung von Richtlinie 4!

Felder dienen der Speicherung von vielen Werten desselben Typs. Über mehrdimensionale Arrays können auch komplexere Strukturen (Paradebeispiel Matrix) nachgebildet werden. Die Elemente eines Feldes werden über einen bei `0` beginnenden Zähler indiziert.

Zum einfachen Durchlaufen und Auslesen eines Feldes kann auch das `foreach`-Konstrukt verwendet werden. Die so erhaltenen Werte können nicht verändert werden!

Variablen kann man mit Hilfe von Qualifizierern mit zusätzlichen Eigenschaften versehen. Hierzu zählt die »konstante Variable«, die nur gelesen werden kann.

Variablen kann man mit Hilfe der Funktion `WriteLine()` ausgeben. Dazu sind in den Ausgabetext Platzhalter einzufügen und die Variablen als Parameter an die Funktion zu übergeben. Über die Tastatur kann man mit Hilfe der Funktion `ReadLine()` Variablenwerte einlesen. Der Rückgabewert dieser Funktion ist eine Zeichenkette; der Wert muss durch eine anschließend durchzuführende Konvertierung gewonnen werden.

3.8 Übungen

Übung 3.1 *

a) Üben Sie die Deklaration und Initialisierung von Variablen, indem Sie kleine Rechenprogramme schreiben. Addieren Sie hierzu einzelne Werte, die Sie einmal in Initialisiererschreibweise deklariert haben, ein andermal durch eine Zuweisung im Programm.

b) Verwenden Sie auch die Hexadezimalschreibweise!

Übung 3.2 **

Was wird ausgegeben?

```
01: ...
02: int a = 4711;
03: Console.WriteLine("{0}", a = 0);
04: ...
```

Übung 3.3 *

Schreiben Sie ein Programm, das die Felder einer 2x2-Matrix mit Hilfe der `ReadLine()`-Funktion einliest und anschließend die Determinante dieser Matrix berechnet. Zur Vereinfachung können Sie davon ausgehen, dass die Determinante existiert und berechnet werden kann. Verwenden Sie hierzu zunächst noch vier einzelne Variablen. Achten Sie auch auf eine saubere Ausgabe!

Übung 3.4 *

Schreiben Sie das Programm aus Übung 3.3 so um, dass anstatt einzelner Variablen ein Array verwendet wird.

Übung 3.5 **

Schreiben Sie ein Programm, das eine Zeichenfolge von der Tastatur einliest und anschließend auf dem Bildschirm wieder ausgibt. Testen Sie mit diesem Programm, ob Sie auch Escapesequenzen eingeben können.