
C#
jetzt lerne ich
Visual C# 2012
Das komplette Starterkit für den erfolgreichen Einstieg
Markt+Technik
DIRK LOUIS
Das perfekte Lernpaket: keine Vorkenntnisse erforderlich
Alles, was Sie brauchen, inklusive: Software, Schulung, Spaß
Mehr als 50 Beispielprojekte sowie Übungen zu jedem Kapitel
ALWAYS LEARNING
PEARSON

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.dnb.de> abrufbar.

10 9 8 7 6 5 4 3 2 1

15 14 13

ISBN 978-3-8272-4820-6 Print; 978-3-86325-601-2 PDF; 978-3-86325-156-7 ePub

Covergestaltung: Thomas Arlt, tarlt@adesso21.net
Lektorat: Thomas Pohlmann
Fachlektorat und Korrektorat: Petra Alm
Herstellung: Martha Kürzl-Harrison, mkuerzl@pearson.de
Satz: text&form GbR, Fürstenfeldbruck
Druck und Verarbeitung: Drukarnia Dimograf, Bielsko-Biala
Printed in Poland

3 C#-Grundkurs: Datenverarbeitung

Sie lernen in diesem Kapitel,

- was Konsolenanwendungen sind und wie man sie erstellt,
- was Variablen, Anweisungen, Datentypen und Operatoren sind,
- was Klassen und Arrays sind,
- welche Bedeutung den Namespaces zukommt.

Im vorangehenden Kapitel haben wir bereits zwei erste Programme aufgesetzt: Windows-Anwendungen mit grafischer Benutzeroberfläche und Steuerelementen. Für all dies mussten wir nicht einmal richtig programmieren – dank der RAD-Umgebung von Visual C# konnten wir uns alles mit der Maus zusammenklicken. Als wir aber daran gingen, eine Ereignisbehandlung für das Drücken des Schalters aufzusetzen, haben wir die Grenzen der RAD-Umgebung kennengelernt. Sie ist wunderbar geeignet, um grafische Benutzeroberflächen zusammenzustellen oder Software-Bausteine, die als Komponenten zur Verfügung stehen, in ein Programm zu integrieren, doch wenn es darum geht, den funktionellen Code aufzusetzen, der hinter der schönen grafischen Oberfläche steht, dann sind wir wieder auf uns allein gestellt. Dann hilft uns keine RAD-Umgebung mehr, dann helfen nur noch gute C#-Kenntnisse.

Die nächsten drei Kapitel sind daher ganz der grundlegenden C#-Syntax gewidmet.

3.1 Konsolenanwendungen

C#-Programme gibt es in verschiedenen Ausprägungen. So bezeichnen wir zum Beispiel Anwendungen, die über eine grafische Benutzeroberfläche (also Fenster, Schalter, etc.) verfügen, als GUI- oder Windows-Anwendungen, wohingegen Programme, die ohne eine solche Benutzeroberfläche auskommen, unter dem Begriff *Konsolenanwendungen* zusammengefasst werden. Die Bezeichnung rührt daher, dass Ein- und Ausgaben nur über eine Konsole[1] erfolgen können.

Reine Konsolenanwendungen werden heutzutage eigentlich nur noch für Aufgaben verwendet, die unsichtbar im Hintergrund erledigt werden können – beispielsweise für die Implementierung von Webservern. In der bunten, Multimedia-verwöhnten Welt der Endanwender spielen Konsolenanwendungen keine große Rolle mehr, da heute jedermann nach Programmen verlangt, die grafische Benutzeroberflächen (GUIs = *Graphical User Interfaces*) haben. Trotzdem werden wir uns in diesem und dem nächsten Kapitel ausschließlich mit Konsolenanwendungen beschäftigen, und zwar aus didaktischen Gründen. Wenn wir uns in der Folge mit der Definition von Variablen, mit Operatoren, Bedingungen und Schleifen oder mit den Grundkonzepten der objektorientierten Programmierung beschäftigen, soll uns nichts von der eigentlichen Thematik ablenken. Darum verzichten wir auf die grafische Oberfläche und konzentrieren uns ganz auf den funktionellen Code.

1 Unter Windows ist dies die Eingabeaufforderung, die Sie unter Windows 7 über Start/Alle Programme/Zubehör aufrufen können. Unter Windows 8 drücken Sie die Tastenkombination ⊞+X und wählen den Befehl Eingabeaufforderung in dem erscheinenden Menü aus.

3.1.1 Das Grundgerüst

Das minimale Grundgerüst einer C#-Anwendung ohne grafische Benutzeroberfläche ist erstaunlich kurz. Tatsächlich braucht man im Grunde ja nichts anderes als eine `Main()`-Methode. Da die Programmausführung, wie Sie aus Kapitel 2 bereits wissen, mit dem Aufruf dieser Methode beginnt, müssen Sie nur den gesamten Programmcode in den Rumpf dieser Methode schreiben.

```
using System;

class HalloWelt
{
   static void Main()
   {
      Console.WriteLine("Hallo Welt!");
   }
}
```

Listing 3.1: Ein minimales Anwendungsgerüst

Listing 3.1 zeigt uns, wie ein minimales ausführbares C#-Programm aussehen könnte. Es besteht aus einer einzigen Klasse, die als einziges Klassenelement die statische Methode `Main()` enthält.

Wenn das C#-Programm gestartet wird, beginnt der JIT-Compiler die Ausführung, indem er die `Main()`-Methode sucht und dort zur ersten Anweisung (in unserem Beispiel also zum `Console.WriteLine()`-Aufruf) springt und diese ausführt. Danach werden schrittweise alle weiteren Anweisungen ausgeführt, bis die schließende Klammer der `Main()`-Methode erreicht wird. Im obigen Miniprogramm gibt es nur eine Anweisung, die den Text »Hallo Welt!« ausgibt.

Hinweis In einem C#-Programm darf es nur eine einzige Klasse mit einer statischen `Main()`-Methode geben.

3.1.2 Konsolenanwendungen in Visual C#

Nun wird es Zeit, dieses Programm mit Visual C# zu erstellen. Zum Glück bietet Visual C# direkte Unterstützung für Konsolenanwendungen, d.h., es gibt unter den von Visual C# vorinstallierten Vorlagen (Aufruf über Datei/Neu) ein Symbol namens Konsolenanwendung, das Ihnen die lästige Arbeit, ein Programmgerüst zu erstellen, abnimmt.

1. Rufen Sie den Menübefehl Datei/Neues Projekt auf, wählen Sie in dem Dialogfeld unter der Kategorie Vorlagen/Visual C# die Vorlage Konsolenanwendung, geben Sie als Programmnamen *HalloWelt* ein, wählen Sie einen SpeicherOrt, achten Sie darauf, dass die Option Projektmappenverzeichnis erstellen deaktiviert ist, und klicken Sie auf OK.

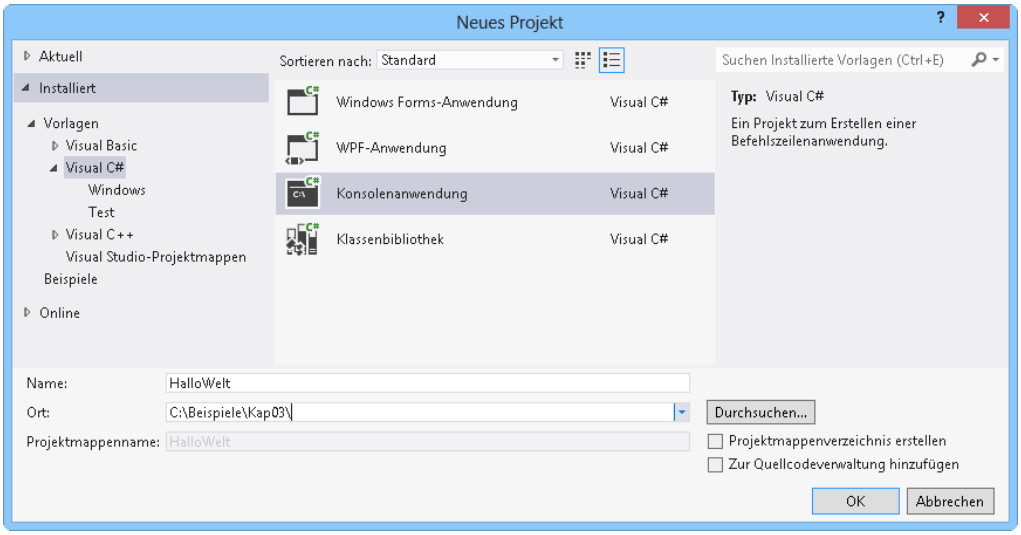

Abbildung 3.1: Eine Konsolenanwendung mit Visual C# anlegen

Im Editor von Visual C# wird das Programmgerüst der Konsolenanwendung, *Program.cs*, angezeigt. Da es sich hierbei um ein noch leeres Gerüst handelt, wollen wir das Programm um eine Codezeile erweitern und den altbekannten Programmierergruß »Hallo Welt!« ausgeben.

2. Fügen Sie dazu in die geschweiften Klammern der `Main()`-Methode folgende Anweisung ein:

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

3. Speichern Sie das Programm (Strg + ⇧ + S).

4. Kompilieren Sie es (F7) und führen Sie es anschließend aus (Strg + F5).

Es öffnet sich das Konsolenfenster mit der von Ihnen vorgegebenen Begrüßung. Durch Drücken einer beliebigen Taste lässt sich das Konsolenfenster wieder schließen.

Achtung Wenn Sie Konsolenanwendungen mit dem Befehl DEBUGGEN/DEBUGGING STARTEN (Tastenkombination F5) ausführen, wird das Konsolenfenster direkt nach Beendigung des Programms geschlossen, sodass unter Umständen keine Zeit bleibt, die Ausgaben des Programms zu begutachten. Um dies zu verhindern, sollten Sie zum Testen Strg + F5 drücken oder als letzte Anweisung in `Main()` die `Console`-Methode `ReadLine()` aufrufen, die das Programm so lange anhält, bis der Anwender die ↵-Taste drückt.

Hinweis Konsolenfenster erscheinen unter Windows üblicherweise mit weißem Text vor schwarzem Hintergrund. Für die Abbildungen in diesem Buch haben wir das Farbschema auf schwarzen Text vor weißem Hintergrund umgestellt. Die betreffenden Einstellungen können übrigens über den EIGENSCHAFTEN-Befehl aus dem Systemmenü des Konsolenfensters aufgerufen werden.

3.1.3 Konsolenanwendungen außerhalb von Visual C# ausführen

Wenn Sie Konsolenanwendungen wie Windows-Anwendungen starten, also etwa durch Doppelklicken im Windows Explorer oder eines zuvor angelegten Desktop-Symbols (vgl. Kapitel 2.5), ergibt sich unter Umständen ein Problem: Sie sehen nichts von dem Programm!

Konsolenanwendungen stammen aus einer Zeit, als es noch keine grafischen Desktops wie unter Windows gab. Damals war der ganze Bildschirm ausgefüllt von einer schwarzen Konsole, auf der der Benutzer Befehle eintippen konnte. Über diese Konsole nahm die Anwendung Eingaben entgegen und auf sie gab die Anwendung ihre Ausgaben aus.

Seitdem haben sich die Zeiten gründlich gewandelt. Wenn Sie heute eine Konsolenanwendung unter Windows aufrufen, fehlt der Anwendung die Konsole. Das Windows-Betriebssystem springt der Anwendung daher zu Hilfe und stellt ihr automatisch ein passendes Konsolenfenster zur Verfügung. Nur leider schließt Windows dieses Konsolenfenster auch automatisch, wenn die Konsolenanwendung beendet ist. Dies führt schnell dazu, dass dem Anwender keine Zeit bleibt, die Ausgaben der Konsolenanwendung zu lesen.

Die Lösung zu diesem Problem ist, zuerst ein Konsolenfenster zu öffnen und dann von diesem aus die Konsolenanwendung zu starten. Dann wird das Konsolenfenster nicht mehr von Windows verwaltet, sondern von Ihnen, und Sie haben alle Zeit der Welt, um die Ausgaben der Anwendung zu lesen oder Sie ein zweites Mal auszuführen oder noch weitere Anwendungen zu starten.

Um die Konsolenanwendung *HalloWelt* von einer Konsole aus zu starten, gehen Sie wie folgt vor:

1. Öffnen Sie ein Konsolenfenster.

   Unter Windows XP/Vista/7 öffnen Sie dazu das Start-Menü und wählen unter Alle Programme\Zubehör den Eintrag Eingabeaufforderung. Unter Windows 8 drücken Sie die Tastenkombination ⊞+X und wählen den Befehl Eingabeaufforderung in dem erscheinenden Menü aus.

2. Wechseln Sie in der Konsole mithilfe des *cd*-Befehls in das Verzeichnis mit der *.exe*-Datei, also beispielsweise:

   ```
   Prompt> cd c:\Beispiele\Kap03\HalloWelt\bin\Debug
   ```

3. Starten Sie die Anwendung über ihren Namen.

   ```
   Prompt> HalloWelt
   ```

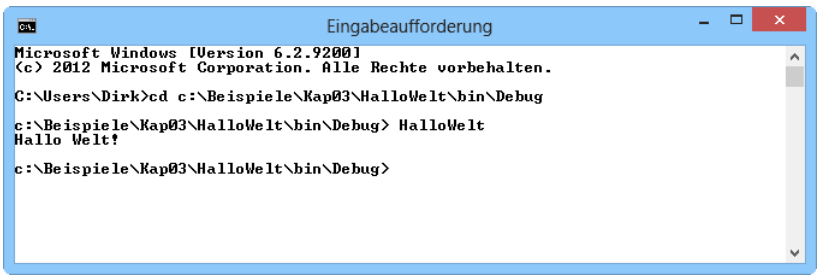

Abbildung 3.2: Ausführung einer Konsolenanwendung

Hinweis Die Konsole ist eine Schnittstelle zum Betriebssystem, über die Sie Zeile für Zeile Befehle abschicken können. Die Eingabe erfolgt in der jeweils untersten Zeile nach dem sogenannten Prompt (der per Voreinstellung meist das aktuelle Verzeichnis angibt).

Jeder Befehl muss durch Drücken der ↵-Taste abgeschickt werden. Wenn sich Ihre Beispiele nicht auf dem Laufwerk `C:\` befinden, wechseln Sie zuerst mit `Laufwerksbuchstabe:` zu dem gewünschten Laufwerk – also beispielsweise `d:`.

Tipp Die Konsole merkt sich die zuletzt eingegebenen Befehle. Wenn Sie also eine Anwendung mehrmals ausführen möchten, müssen Sie dazu den Namen der Anwendung nicht jedes Mal neu eingeben, sondern können mithilfe der Pfeiltasten durch die Konsolen-History laufen, bis der Name wieder in der Eingabezeile steht. (Mehr Informationen zur Bedienung der Konsole finden Sie als Tutorial zum Herunterladen auf der Website *www.carpelibrum.de*.)

3.2 Datentypen und Variablen

Die Aufgabe eines jeden Computerprogramms ist die Verarbeitung von irgendwelchen Informationen, die im Computerjargon meist Daten genannt werden. Das können Zahlen sein, aber auch Buchstaben, ganze Texte oder Bilder und Zeichnungen. Dem Rechner ist diese Unterscheidung gleich, da er letztlich alle Daten in Form von endlosen Zahlenkolonnen in Binärdarstellung (nur Nullen und Einsen) verarbeitet.

Stellen Sie sich vor, wir schreiben das Jahr 1960 und Sie sind stolzer Besitzer einer Rechenmaschine, die Zahlen und Text verarbeiten kann. Beides allerdings in Binärformat. Um Ihre Freunde zu beeindrucken, lassen Sie den »Computer« eine kleine Subtraktion berechnen, sagen wir:

```
8754 – 398 = ?
```

Zuerst rechnen Sie die beiden Zahlen durch fortgesetzte Division mit 2 ins Binärsystem um (wobei die nicht teilbaren Reste der aufeinander folgenden Divisionen, von rechts nach links geschrieben, die gewünschte Binärzahl ergeben).

```
10001000110010 – 110001110 = ?
```

Die Binärzahlen stanzen Sie sodann als Lochkarte und lassen diese von Ihrem Computer einlesen. Dann drücken Sie noch die Taste für Subtraktion und ohne Verzögerung erscheint das korrekte Ergebnis:

```
10000010100100
```

Zweifelsohne werden Ihre Freunde von dieser Maschine äußerst beeindruckt sein. Trotzdem lässt sich nicht leugnen, dass die Interpretation der Binärzahlen etwas unhandlich ist, und zwar erst recht, wenn man neben einfachen ganzen Zahlen auch Gleitkommazahlen, Texte und Bitmaps in Binärformat speichert.

Für die Anwender von Computern ist dies natürlich nicht zumutbar und die Computer-Revolution – die vierte der großen Revolutionen (nach der Glorious Revolution, England 1688, der Französischen Revolution von 1789 und der Oktoberrevolution, 1917 in Russland) – konnte nur stattfinden, weil man einen Ausweg fand. Dieser bestand einfach darin, es der Software – dem laufenden Programm – zu überlassen, die vom Anwender eingegebenen Daten (seien es Zahlen, Text, Bilder etc.) ins Binärformat umzuwandeln und umgekehrt die auszugebenden Daten wieder vom Binärformat in eine leicht lesbare Form zu verwandeln.

»Gemeinheit«, werden Sie aufbegehren, »da wurde das Problem ja nur vom Anwender auf den Programmierer abgewälzt«. Ganz so schlimm ist es nicht. Der C#-Compiler nimmt uns hier das Gröbste ab. Alles, was wir zu tun haben, ist, dem Compiler anzugeben, mit welchen Daten wir arbeiten möchten und welchem Datentyp diese Daten angehören (sprich, ob es sich um Zahlen, Text oder Sonstiges handelt).

Betrachten wir ein Beispiel:

```
static void Main(string[] args)
{
    int ersteZahl;
    int zweiteZahl;
    int ergebnis;

    ersteZahl = 8754;
    zweiteZahl = 398;
}
```

Hinweis Obiger Quelltextauszug stellt kein vollständiges Programm dar – es fehlt die umliegende Klasse, in der die `Main()`-Methode definiert sein muss.

In der `Main()`-Methode werden zuerst die benötigten Variablen definiert.

Variablen

Die Variablen eines Programms sind nicht mit den Variablen mathematischer Berechnungen gleichzusetzen. *Variablen* bezeichnen Speicherbereiche im RAM (Arbeitsspeicher), in denen ein Programm Werte ablegen kann. Um also mit Daten arbeiten zu können, müssen Sie zuerst eine Variable für diese Daten definieren. Der Compiler sorgt dann dafür, dass bei Ausführung des Programms Arbeitsspeicher für die Variable reserviert wird. Für den Compiler ist der Variablenname einfach ein Verweis auf den Anfang eines Speicherbereichs. Als Programmierer identifiziert man eine Variable mehr mit dem Wert, der gerade in dem zugehörigen Speicherbereich abgelegt ist.

Bei der *Definition* geben Sie nicht nur den Namen der Variablen an, sondern auch deren Datentyp. Dieser Datentyp verrät dem Compiler, wie der Inhalt des Speicherbereichs der Variablen zu interpretieren ist. Im obigen Beispiel benutzen wir den Datentyp `int`, der für einfache Ganzzahlen steht.

Merksatz Zu jeder Variablendefinition gehört auch die Angabe eines *Datentyps*. Dieser gibt dem Compiler an, wie der Speicherinhalt der Variablen zu interpretieren ist.

```
int ersteZahl;
```

Dem Konzept der Datentypen verdanken wir es, dass wir der Variablen `ersteZahl` eine Ganzzahl zuweisen können und nicht etwa wie im obigen Beispiel des Lochkartenrechners die Dezimalzahl in Binärcode umrechnen müssen.

```
ersteZahl = 8754;
```

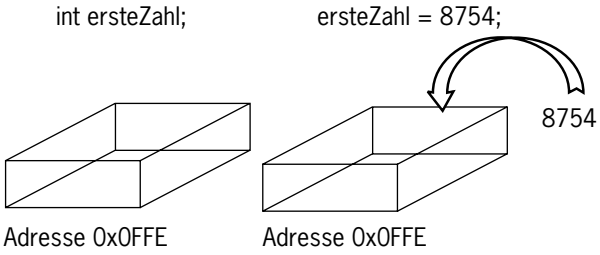

Abbildung 3.3: Definition und Zuweisung

3.2.1 Der »Wert« der Variablen

Wenn eine Variable einen Speicherbereich bezeichnet, dann ist der Wert einer Variablen der interpretierte Inhalt dieses Speicherbereichs. Im obigen Beispiel ist der Wert der Variablen `ersteZahl` nach der Anweisung

```
ersteZahl = 8754;
```

also `8754`. Wenn Sie der Variablen danach einen anderen Wert zuweisen, beispielsweise

```
ersteZahl = 5;
```

ist der Wert in der Folge gleich `5`.

Was die Variablen für den Programmierer aber so wertvoll macht, ist, dass er sich nicht mehr um die Speicherverwaltung zu kümmern braucht. Es ist zwar von Vorteil, wenn man weiß, dass hinter einer Variablen ein Speicherbereich steht, für die tägliche Programmierarbeit ist es aber meist nicht erforderlich. Wir sprechen nicht davon, dass wir mithilfe des Variablennamens einen eindeutig bezeichneten Platz im Arbeitsspeicher referenzieren und in diesen einen Wert schreiben; wir sagen einfach, dass wir der Variablen einen Wert zuweisen. Wir sprechen nicht davon, dass das interpretierte Bitmuster in dem Speicherbereich der `int`-Variablen `ersteZahl` gleich `5` ist; wir sagen einfach, `ersteZahl` ist gleich `5`. Wir sprechen nicht davon, dass wir mithilfe des Variablennamens einen eindeutig bezeichneten Platz im Arbeitsspeicher referenzieren und dessen Wert auslesen; wir sagen einfach, dass wir den Wert der Variablen auslesen.

3.2.2 Variablen bei der Definition initialisieren

Wenn Sie möchten, können Sie einer Variablen auch direkt bei der Definition einen Anfangswert zuweisen. In einem solchen Falle spricht man von *Initialisierung*.

```
static void Main(string[] args)
{
    int ersteZahl = 8754;
    int zweiteZahl = 398;
    int ergebnis;
}
```

3.2.3 Werte von Variablen abfragen

Einen Wert in einer Variablen abzuspeichern, ist natürlich nur dann interessant, wenn man auf den Wert der Variablen später noch einmal zugreifen möchte – beispielsweise um den Wert in eine Formel einzubauen oder auszugeben.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Variablen
{
    class Program
    {
        static void Main(string[] args)
        {
            int ersteZahl = 8754;
            int zweiteZahl = 398;
            int ergebnis;

            ergebnis = ersteZahl - zweiteZahl;

            Console.WriteLine("8754 - 398  = " + ergebnis);
        }
    }
}
```

Listing 3.2: Programmieren mit Variablen (aus dem Projekt Variablen)

Der hier verwendete Aufruf von `WriteLine()` bedarf einer Erklärung. Unsere Intention ist es, die Ausgabe

```
8754 – 398 = 8356
```

zu erzeugen. Die Aufgabenstellung bis zum Gleichheitszeichen geben wir als String vor, das Ergebnis soll aus dem Wert der Variablen `ergebnis` übernommen werden. Hierfür gibt es verschiedene Möglichkeiten.

Eine Möglichkeit ist, wie im Beispiel, die Ausgabe mithilfe des +-Operators aus Teilstring und Variable zusammenzusetzen:

```
Console.WriteLine("8754 - 398  = " + ergebnis);
```

Eine andere Möglichkeit ist, in den auszugebenden String einen nummerierten Platzhalter einzufügen und der `WriteLine()`-Methode hinter dem String die Variable zu übergeben, durch deren Wert der Platzhalter ersetzt werden soll (mehr dazu in Kapitel 7.1).

Achtung Variablen, die Sie innerhalb einer Methode definieren, sind nur in der Methode und nur in den nachfolgenden Anweisungen gültig.

3.2.4 Die elementaren Datentypen

Zu jeder Variablendefinition gehört die Angabe des Datentyps, der dem Compiler mitteilt, welche Werte der Variablen zugewiesen werden können. Neben dem Datentyp `int` für Ganzzahlen kennt C# noch eine Reihe weiterer einfacher Datentypen:

Datentyp	Beschreibung	Werte
`bool`	boolescher Wert (wahr, falsch)	`true`, `false`
`char`	Zeichen, Buchstabe	`'a'`, `'2'` Unicode-Werte wie `\u00A3` (£)
`byte`	ganze Zahl	0 bis +255
`short`	ganze Zahl	-32.768 bis 32.767
`int`	ganze Zahl	-2.147.483.648 bis +2.147.483.647
`long`	ganze Zahl	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
`float`	Gleitkommazahl	$1{,}5 \times 10^{-45}$ bis $3{,}4 \times 10^{38}$
`double`[1]	Gleitkommazahl	$5{,}0 \times 10^{-324}$ bis $1{,}7 \times 10^{308}$

Tabelle 3.1: Elementare Datentypen

1 Für Gleitkommazahlen (Zahlen mit Nachkommaanteil) sollten Sie generell den Datentyp `double` verwenden. Mit diesem lassen sich nicht nur größere Werte darstellen, die Zahlen lassen sich zudem in größerer Präzision (mit mehr Nachkommastellen) darstellen.

Unicode

In der Tabelle ist für `char`-Variablen der Unicode angegeben. *Unicode* ist ein standardisierter Zeichensatz mit 65.536 Zeichen, mit dem alle diversen Umlaute und Sonderzeichen aller gängigen Sprachen, ja sogar japanische und chinesische Schriftzeichen, dargestellt werden können!

Wie Sie sehen, gibt es verschiedene Datentypen mit unterschiedlichen Wertebereichen. Um zum Beispiel eine ganze Zahl abzuspeichern, haben Sie die Wahl zwischen `byte`, `short`, `int` und `long`! Die größeren Wertebereiche erkauft man sich mit einem höheren Speicherverbrauch. Eine `long`-Variable benötigt beispielsweise doppelt so viel Speicher wie eine `int`-Variable. Glücklicherweise ist Arbeitsspeicher kein allzu großes Problem mehr und viele Programmierer verwenden standardmäßig `int` oder `long` für ganzzahlige Werte und `double` für Gleitkommazahlen.

Hinweis Der *Datentyp* legt also nicht nur fest, wie der Wert der Variablen zu interpretieren ist, er gibt auch an, wie groß der für die Variable bereitzustellende Speicherbereich sein muss.

Hier einige Beispiele:

```
int ganzeZahl;
double krummeZahl;
bool antwort;
short klein = -4;
char buchstabe = 'Ü';
char ziffer = '4';
```

Es ist auch möglich, mehrere Variablen des gleichen Typs durch Komma getrennt auf einmal zu definieren:

```
bool ja, nein, oder_doch;
```

3.2.5 Strings

Strings haben wir bisher fast ausschließlich als String-Konstanten verwendet – beispielsweise in den Aufrufen von `Console.WriteLine()`:

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

String-Konstanten

Wie Sie bereits gesehen haben, werden String-Konstanten dadurch gekennzeichnet, dass sie mit doppelten Anführungszeichen beginnen und mit doppelten Anführungszeichen enden.

Die Frage ist nur, was passiert, wenn ein String selbst doppelte Anführungszeichen enthält.

```
"Der Pfarrer sprach: "Der Tod ist der Sünde Sold.""
```

Wenn Sie versuchen, einen solchen String auszugeben oder anderweitig zu verarbeiten, werden Sie dafür vom Compiler einen Haufen wütender Fehlermeldungen empfangen. Der Grund ist, dass der Compiler den String nicht nach dem letzten, sondern direkt nach dem zweiten doppelten Anführungszeichen für beendet ansieht. Aus Sicht des Compilers besteht die obige Zeile aus dem String `"Der Pfarrer sprach: "`, gefolgt von den Zeichen `Der Tod ist der Sünde Sold.`, mit denen er überhaupt nichts anfangen kann, und schließlich einem zweiten, leeren String `""`.

Um dennoch Strings mit doppelten Anführungszeichen definieren und verarbeiten zu können, bräuchte man einen Weg, wie man dem Compiler anzeigen kann, dass das folgende doppelte Anführungszeichen eben nicht das Ende des Strings signalisiert, sondern als normales Zeichen zu behandeln ist. Diese Möglichkeit eröffnet uns das Escape-Zeichen.

Escape-Sequenzen

Mithilfe des *Escape-Zeichens* \ kann man zum einen Zeichen, die für den Compiler eine besondere Bedeutung haben (" oder \), in Strings als einfache Textzeichen kennzeichnen (\" oder \\), zum anderen kann man bestimmte Sonderzeichen in einen Text einfügen (beispielsweise `\t` zum Einfügen eines Tabulators).

Escape-Sequenz	Beschreibung
`\'`	Einfaches Anführungszeichen
`\"`	Doppeltes Anführungszeichen
`\\`	Backslash
`\0`	Null-Zeichen
`\a`	Signalton
`\b`	Rückschritttaste
`\f`	Seitenvorschub
`\n`	Neue Zeile (Zeilenumbruch)
`\t`	Horizontaler Tabulator
`\v`	Vertikaler Tabulator

Tabelle 3.2: Die Escape-Sequenzen

Um beispielsweise doppelte Anführungszeichen in einem String zu verwenden, brauchen Sie den Anführungszeichen im String lediglich das Escape-Zeichen \ voranzustellen:

```
"Der Pfarrer sprach: \"Der Tod ist der Sünde Sold.\""
```

Mithilfe der Escape-Sequenzen `\n` und `\t` können Sie Textausgaben übersichtlicher gestalten:

```
using System;
using System.Collections.Generic;
using System.Text;
```

```
namespace Escapezeichen
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine();
            Console.WriteLine(" Daten von Dirk Louis: \n");
            Console.WriteLine("\t Alter     : 39");
            Console.WriteLine("\t Groesse   : 178 cm");
            Console.WriteLine("\t Gewicht   : 77 kg");
            Console.WriteLine("\n");
        }
    }
}
```

Listing 3.3: Umbrüche und Tabulatoren in Strings (aus dem Projekt Escapezeichen)

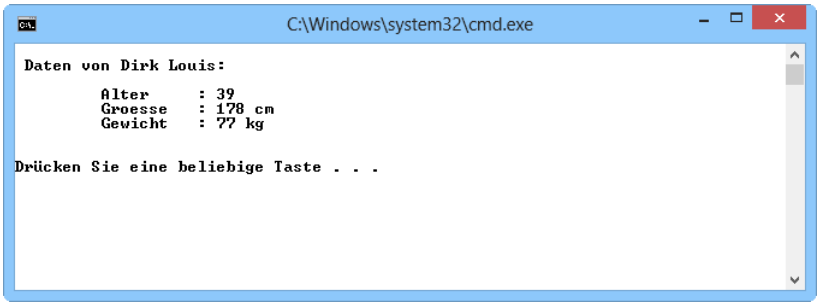

Abbildung 3.4: Mithilfe von Tabulatoren, Leerzeichen und Umbrüchen formatierte Ausgabe

Hinweis Beachten Sie, dass die Doppelpunkte in den Zeilen durch Leerzeichen untereinander ausgerichtet wurden. Würde man hinter den einzelnen Eigenschaften Tabulatoren einfügen, wäre nicht sichergestellt, dass die Doppelpunkte untereinander stehen. Dies liegt daran, dass Tabulatoren bis zu vorgegebenen Zeichenspalten vorrücken. Wenn das Ende von `Alter` zufällig vor und das Ende von `Groesse` und `Gewicht` hinter einer solchen Spalte liegen, springen die Tabulatoren hinter den Wörtern zu unterschiedlichen Spalten.

Zeilenumbrüche und Aneinanderreihung von Strings

Manchmal müssen Sie mit Strings arbeiten, die so lang sind, dass Sie sie am liebsten auf mehrere Zeilen verteilen würden.

```
Console.WriteLine("Dies ist ein langer Text, den man im Quelltext am liebsten auf mehrere
Zeilen verteilen würde!");
```

Dieser Text ist nur für den Ausdruck in diesem Buch umgebrochen worden. Im Quelltext eines Programms würde er in einer Zeile stehen – mit dem Effekt, dass das Ende des Textes ohne Scrollen nicht mehr zu lesen ist.

Wenn Sie aber im Quelltext einen Umbruch in eine String-Konstante einfügen, ernten Sie dafür später beim Kompilieren eine Fehlermeldung, denn Umbrüche in String-Konstanten sind nicht erlaubt.

Die Lösung zu diesem Problem besteht darin, den String in zwei oder mehrere Teilstrings aufzubrechen und diese mithilfe des Operators + wieder aneinanderzuhängen. Die einzelnen Teilstrings können Sie dann auf mehrere Zeilen verteilen:

```
Console.WriteLine("Dies ist ein langer Text, den man"
                + "im Quelltext am liebsten auf mehrere"
                + "Zeilen verteilen würde!");
```

Achtung Den +-Operator für Strings nennt man auch Verkettungs- oder *Konkatenationsoperator*. Er ist der einzige der arithmetischen Operatoren, der auch für Strings definiert ist.

String-Variablen

Mithilfe des Datentyps `string` (ein Synonym für die Klasse `System.String`) können Sie Variablen für Strings definieren.

```
string meinStr;
```

Diesen können Sie String-Konstanten oder Strings aus anderen `string`-Variablen zuweisen.

```
string str1 = "Hallo Welt!";
string str2;

str2 = str1;
```

Strings vergleichen

Um festzustellen, ob zwei Strings gleich oder ungleich sind, können Sie die Operatoren `==` und `!=` verwenden:

```
bool vergleich;
string str1 = "Genau";
string str2 = "Genua";
vergleich = (str1 == str2);      // false
vergleich = (str1 != str2);      // true
```

Zusätzlich gibt es in der Klasse `String` spezielle Methoden zum Vergleichen.

```
int String.Compare( str1, str2 )

int String.CompareOrdinal( str1, str2 )
```

Beide Methoden vergleichen `str1` mit `str2`. Der Rückgabewert zeigt an, ob `str1` kleiner, gleich oder größer als `str2` ist. Die Methode `Compare()` berücksichtigt beim Vergleichen allerdings nationale Eigenheiten (abhängig von der landesspezifischen Einstellung des Computers), die Methode `CompareOrdinal()` berücksichtigt solche Eigenheiten nicht.

Rückgabewert	Bedeutung
-1	`str1` kleiner `str2`
0	`str1` gleich `str2`
1	`str1` größer `str2`

Tabelle 3.3: Rückgabewerte der Vergleichsmethoden

```
int vergleich;
string str1 = "Genau";
string str2 = "Genua";
vergleich = String.Compare( str1, str2 );      // -1
```

Strings vergleichen

Strings werden lexikografisch verglichen, d.h., der Compiler geht die beiden Strings von vorne nach hinten durch und vergleicht Zeichen für Zeichen, bis er auf das erste Zeichen trifft, das in beiden Strings unterschiedlich ist. Dann stellt er fest, welches dieser Zeichen im Alphabet zuerst kommt. Der String, zu dem dieses Zeichen gehört, ist der lexikografisch kleinere der beiden verglichenen Strings.

Sind die beiden Strings bis zum Ende eines Strings gleich, entscheidet die String-Länge. Ist auch diese gleich, sind die beiden Strings gleich.

Weitere Methoden der String-Klasse

Die Klasse `String` stellt noch eine Reihe weiterer interessanter Methoden und Eigenschaften zur Verfügung.

Methode/Eigenschaft	Beschreibung
`int` **`Length`**	Die Länge des Strings (sprich die Anzahl Zeichen im String).

`string` **`String.Copy`**`(str)`	Kopiert den String `str` und liefert die Kopie als Ergebniswert zurück.
	`string str1 = "Hallo Programmierer!";`
	`string str2;`
	`str2 = String.Copy(str1);`

Methode/Eigenschaft	Beschreibung

`int str1.IndexOf(str2)`	Sucht nach dem ersten Vorkommen von `str2` in `str1` und liefert die gefundene Position als `int`-Wert zurück bzw. `-1`, falls der String `str2` nicht in `str1` vorkommt.
	`string str = "123456";`
	`int pos;`
	
	`pos = str.IndexOf("3");  // pos = 2`

`int str1.Insert(`
`startPos,`
`str2)`
	Fügt den String `str2` an der Position `startPos` in den String `str1` ein.
	`string str = "123456";`
	`str = str.Insert(3,"abcd");`
	
	`// str enthält jetzt "123abcd456"`

`string str.Remove(`
`startPos,`
`anzahl)`
	Löscht ab der Position `startPos` genau `anzahl` Zeichen aus dem String.
	`string str = "123abcd456";`
	`str = str.Remove(3,4);  // str = "123456"`
	Achten Sie darauf, nicht über das Ende des Strings hinaus Zeichen zu löschen.

`string str.Replace(`
`alterTeilstring,`
`neuerTeilstring)`
	Ersetzt alle Vorkommen eines Teilstrings durch einen anderen Teilstring und liefert das Ergebnis als Kopie zurück.
	`string str = "Hallo XXXXXXXX!";`
	`str = str.Replace('X', '_');`

`string str.Substring(`
`pos,`
`laenge)`
	Liefert einen Teil des Strings als Kopie zurück. Der Teilstring beginnt bei Position `pos` und geht bis zum Ende des Strings oder hat `laenge` Zeichen.
	`string str1 = "Hallo Programmierer!";`
	`string str2;`
	
	`str2 = str1.Substring(6);`
	
	`// str2 enthält jetzt "Programmierer"`

`string str.ToLower()`
`string str.ToUpper()`
	Methoden zur Umwandlung in Klein- oder Großbuchstaben. Der umgewandelte String wird als Ergebnis zurückgeliefert.
	`string str = "Hallo Programmierer!";`
	`str = str.ToUpper();`
	
	`// str enthält "HALLO PROGRAMMIERER!"`

Methode/Eigenschaft	Beschreibung

`string str.`**`Trim`**`()`	Entfernt sogenannten »Whitespace« vom Anfang und Ende eines Strings und liefert das Ergebnis als neuen String zurück.
	`string str = "\n\t  Hallo Programmierer! \n\t";`
	`str = str.Trim();`
	`Console.WriteLine(str);`
	Ausgabe
	`  Hallo Programmierer!`
	Hinweis: Als *Whitespace* bezeichnet man Zeichen, die im Ausdruck nur als Leerraum zu sehen sind. Dies sind vor allem Leerzeichen, Tabulator und Zeilenumbruch.

`string[] str.`
`Split``(trennzeichen)`
	Diese Methode arbeitet mit Arrays von Strings und einzelnen Zeichen. Arrays werden wir zwar erst weiter unten in Abschnitt 3.6 besprechen, doch die `String`-Methode `Split()` ist zu interessant, um sie hier zu übergehen. Mit ihrer Hilfe kann man nämlich auf elegante Weise einen String in Teilstrings zerlegen. Sie müssen der Methode dazu nur angeben, welches Zeichen oder welche Zeichen als Trennzeichen zu interpretieren sind.
	Der folgende Code-Auszug zerlegt einen Satz in einzelne Wörter:
	`string str = "Zerlege mich in Wörter!";`
	`string[] woerter;`
	`char[]   trennzeichen = {' '};`
	`woerter = str.Split(trennzeichen);`
	`foreach (string wort in woerter)`
	`  Console.WriteLine(wort);`
	Ausgabe
	`Zerlege`
	`mich`
	`in`
	`Wörter!`

Tabelle 3.4: Interessante Methoden der Klasse String

3.2.6 Typumwandlung

C# ist eine sehr typenstrenge Programmiersprache, die sehr darauf achtet, dass eine Variable nur Werte ihres Typs zugewiesen bekommt und auch nur entsprechend ihres Typs verwendet wird. Dies heißt jedoch nicht, dass der Datenaustausch zwischen Variablen unterschiedlicher Typen ganz und gar unmöglich wäre. Ja, es wäre geradezu fatal, wenn die Sprache keine Möglichkeiten zur Typumwandlung kennen würde.

Betrachten wir dazu noch einmal das Programm *Variablen.cs* aus Listing 3.2:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Variablen
{
    class Program
    {
        static void Main(string[] args)
        {
            int ersteZahl = 8754;
            int zweiteZahl = 398;
            int ergebnis;

            ergebnis = ersteZahl - zweiteZahl;

            Console.WriteLine("8754 - 398  = " + ergebnis);
        }
    }
}
```

In der letzten Zeile wird der Wert der `int`-Variablen `ergebnis` auf die Konsole ausgegeben. Auf die Konsole kann man aber nur Strings ausgeben! Des Rätsels Lösung ist, dass die Methode `Console.WriteLine()` den Wert von `ergebnis`, in unserem Beispiel die Zahl 8356, in den String `"8356"` umwandelt. Dieser wird dann von der Methode ausgegeben.

Was die Methode `Console.WriteLine()` kann, würden wir auch gerne können. Die Frage ist nur wie? Wenn Sie nämlich versuchen, eine Zahl an eine `string`-Variable zuzuweisen:

```
string str;
str  = 8754;
```

ernten Sie vom Compiler lediglich eine Fehlermeldung der Form

Eine implizite Konvertierung vom Typ "int" in "string" ist nicht möglich.

Was ist zu tun?

Die Lösung bringt eine spezielle Klasse namens `Convert`. Diese definiert eine Reihe von Methoden, mit denen man die Werte elementarer Datentypen in andere elementare Datentypen umwandeln kann.

Methode	Umwandlung in
ToBoolean()	bool
ToByte()	byte
ToChar()	char
ToDecimal()	decimal
ToDouble()	double
ToInt16()	short
ToInt32()	int
ToInt64()	long
ToString()	string

Tabelle 3.5: Auswahl der wichtigsten Umwandlungsmethoden

Zusätzlich besitzen die elementaren Zahlentypen eine Methode `ToString()`, die den Wert der Zahl als String zurückliefert.

Mithilfe dieser Methoden können Sie Konstanten und die Werte von Variablen in andere Datentypen umwandeln:

```
int    zahl  = 8754;
string str;

str  = 8754.ToString();
str  = zahl.ToString();
```

Doch Vorsicht! Zwar können Sie mithilfe dieser Methoden praktisch beliebige Umwandlungen erzwingen, doch heißt dies nicht, dass diese Umwandlungen auch immer möglich sind. Sehen wir uns hierzu einige Beispiele an.

Umwandlung von Zahlen in Strings

```
static void Main(string[] args)
{
    double zahl = -8754.3;
    string str;

    str = zahl.ToString();

    Console.WriteLine(str);
}
```

Was halten Sie von diesem Programm? Wird die gewünschte Umwandlung erfolgreich sein? Sie wird! Testen Sie das Programm ruhig aus.

Die Umwandlung ist für den Compiler überhaupt kein Problem; er braucht nur den Zahlenwert (-8754.3) in die Zeichenfolge (`"-8754.3"`) zu verwandeln – ganz wie es bei der Übergabe einer Zahlen-Variablen an `Console.WriteLine()` geschieht.

Umwandlung von Strings in Zahlen – Teil 1

```
static void Main(string[] args)
{
    int zahl;
    string str = "123";

    zahl = Convert.ToInt32(str);

    Console.WriteLine(zahl);
}
```

Was halten Sie von diesem Programm?

Auch diese Umwandlung ist unkritisch. In dem String steht die korrekte String-Repräsentation einer Zahl, die der Compiler ohne Probleme in einen Ganzzahlenwert umwandeln kann.

Umwandlung von Strings in Zahlen – Teil 2

```
static void Main(string[] args)
{
    double zahl;
    string eingabe;

    Console.WriteLine("Geben Sie bitte eine Zahl ein: ");
    eingabe = Console.ReadLine();

    zahl = Convert.ToDouble(eingabe);

    Console.WriteLine(zahl);
}
```

Und wie sieht es hiermit aus?

In diesem Programm wird der Anwender aufgefordert, eine Zahl einzugeben. Die Eingabe nimmt das Programm mithilfe der Methode `Console.ReadLine()` entgegen und speichert sie in der `string`-Variablen `eingabe`.

> **Hinweis** Tastatureingaben sind immer Strings. Die Methode `Console.ReadLine()` wandelt diese nicht automatisch um, sondern liefert die gesamte Eingabezeile unverändert als String zurück.

Danach wird die Eingabe mithilfe der Methode `Convert.ToDouble()` in eine Zahl umgewandelt und in `zahl` abgespeichert. Wir verwenden hier `ToDouble()` statt `ToInt32()`, damit der Anwender sowohl Ganzzahlen als auch Zahlen mit Nachkommastellen eingeben kann.

Die Frage ist nun, ob die Umwandlung der `eingabe` in einen `double`-Wert glücken wird?

Tatsächlich lässt sich diese Frage nicht beantworten, da das Ergebnis der Umwandlung nicht vorhersehbar ist – es hängt davon ab, was der Anwender als Eingabe eintippt.

Tippt der Anwender als Zahl beispielsweise 123 oder -3,54 ein, wird die Umwandlung glücken.

Tippt der Anwender eine Gleitkommazahl in einem von C# nicht akzeptierten Format oder gar einen Text wie »Meine Zahl lautet 123« ein, wird die Umwandlung nicht glücken. Die Laufzeitumgebung wird eine Fehlermeldung ausgeben und das Programm wird vorzeitig beendet.

Programmabbrüche durch Benutzereingaben vermeiden

Dass Anwender falsche Eingaben an ein Programm schicken (etwa Text statt Zahlen), ist nicht schön, aber leider unvermeidbar. Sie müssen immer damit rechnen, dass ein Anwender, den Sie auffordern, eine Zahl zwischen 1 und 10 einzugeben, »sieben« eintippt.

Mindestens ebenso unschön ist es aber, wenn Ihr Programm bei solchen Gelegenheiten gleich ganz außer Kontrolle gerät und sich sang- und klanglos verabschiedet. Sie können dies verhindern, indem Sie die Typumwandlung in Ausnahmebehandlungscode einfassen, siehe Kapitel 4.3.

Umwandlung von Gleitkommazahlen in Ganzzahlen

```
static void Main(string[] args)
{
    double zahl1 = 1234.567;
    int zahl2;

    zahl2 = Convert.ToInt32(zahl1);

    Console.WriteLine(zahl2);
}
```

Neues Spiel, neues Glück! Wie sieht es mit dieser Typumwandlung aus?

Hier wird eine Gleitkommazahl mit einem Nachkommaanteil in eine Ganzzahl ohne Nachkommastellen umgewandelt. Dies ist meist unproblematisch (solange die Gleitkommazahl vom Betrag her nicht so groß ist, dass sie außerhalb des Wertebereichs des Ganzzahltyps liegt). Allerdings wird der Nachkommaanteil bei der Umwandlung gerundet. Wenn Sie dies nicht stört, können Sie eine solche Umwandlung ohne Probleme vornehmen.

Für die Umwandlung zwischen den elementaren Datentypen für Ganz- und Gleitkommazahlen gibt es übrigens noch zwei andere Verfahren zur Typumwandlung.

Wenn der Wertebereich des Quelldatentyps eine Teilmenge des Wertebereichs des Zieldatentyps darstellt, können Sie den Wert direkt zuweisen – der Compiler führt die Typumwandlung automatisch durch (implizite Typumwandlung).

```
short  s_zahl = 110;
int    i_zahl = 220;
double d_zahl = 330;

i_zahl = s_zahl;
d_zahl = i_zahl;
```

Wenn der Wertebereich des Quelldatentyps größer als der Wertebereich des Zieldatentyps ist, können Sie den Wert durch Voranstellung des Zieltyps umwandeln (explizite Typumwandlung).

```
short  s_zahl = 110;
int    i_zahl = 220;
double d_zahl = 330;

i_zahl = (int) d_zahl;
s_zahl = (short) i_zahl;
```

Hinweis Bei der impliziten oder expliziten Umwandlung eines `double`-Werts in einen `int`-Wert wird nicht gerundet (wie es `Convert` macht). Stattdessen wird der Nachkommaanteil einfach gelöscht.

Hinweis Die explizite Typumwandlung mit dem `()`-Operator wird in der Programmierung auch als *Casting* bezeichnet.

3.2.7 C# für Pedanten

Auch wenn Ihnen die Syntax von C# einerseits viele Möglichkeiten offen lässt, ist sie andererseits doch recht starr vorgegeben und der Compiler wacht penibel darüber, dass Sie sich an die korrekte Syntax halten.

Wenn Sie es sich also nicht mit dem Compiler verderben möchten, sollten Sie insbesondere auf folgende Punkte achten:

- Alle *Anweisungen* (also Zuweisungen, Methodenaufrufe und Variablendefinitionen) müssen mit einem Semikolon abgeschlossen werden.

  ```
  krummeZahl = 47.11;
  ```

- C# unterscheidet streng zwischen *Groß- und Kleinschreibung*. Wenn Sie also eine Variable namens `krummeZahl` definiert haben, dann müssen Sie auch `krummeZahl` schreiben, wenn Sie auf die Variable zugreifen möchten, und nicht `krummezahl`, `KrummeZahl` oder `KRUMMEZAHL`.

Und natürlich gibt es auch Regeln für die Einführung von Bezeichnern, also beispielsweise Variablennamen. Wir wollen uns an dieser Stelle aber nicht mit den endlosen Vorschriften befassen, welche die Konstruktion von Variablennamen regeln. Merken Sie sich einfach Folgendes:

- *Variablennamen* können beliebig lang sein, müssen mit einem Buchstaben oder einem Unterstrich '_' beginnen und dürfen nicht identisch mit einem Schlüsselwort der Sprache sein.

Sie dürfen also Ihre Variable nicht `class` nennen, da dies ein reserviertes Wort, ein sogenanntes *Schlüsselwort*, ist. Im Anhang zur C#-Syntax finden Sie eine Liste der C#-Schlüsselwörter. Neben Buchstaben und Ziffern sind fast alle Unicode-Zeichen erlaubt. Insbesondere sind Umlaute erlaubt. Wenn Sie

also eine Variable `Begrüßung` nennen möchten, brauchen Sie sie nicht wie in anderen Programmiersprachen als `Begruessung` zu definieren, sondern können ruhig `Begrüßung` schreiben.

> **Hinweis** Ob man allerdings von dieser Option auch Gebrauch machen sollte, ist eine andere Frage. International ist das Englische die Sprache der Programmierer, und auch wenn Sie nicht unbedingt gleich dazu übergehen müssen, Ihren Klassen, Methoden und Variablen englische Namen zu geben, so empfiehlt es sich in Hinblick auf den weltweiten Austausch von Code doch, in Bezeichnern nur den englischen Zeichensatz zu verwenden (also das romanische Alphabet ohne Umlaute).

3.3 Variablen kontra Konstanten

Muss man wirklich erst erwähnen, dass man den Wert einer Variablen ändern kann (indem man ihr einen neuen Wert zuweist), wohingegen der Wert einer Konstanten unveränderlich ist? Wohl nicht. Interessanter ist es schon zu erfahren, wie man mit Konstanten arbeitet. Dazu gibt es zwei Möglichkeiten:

Erstens: Sie tippen die Konstante direkt als Wert ein. Man spricht dann von sogenannten *Literalen*.

```
krummeZahl = 47.11;                    // Zuweisung eines Literals

krummeZahl = ganzeZahl + 47.11;        // Addition des Werts aus der
                                       // Variablen ganzeZahl und
                                       // des Literals 47.11
```

Da mit einem Literal kein Datentyp verbunden ist, muss der Compiler den Datentyp aus der Syntax des Literals ablesen:

Datentyp	Literal
`bool`	`true`, `false`
`char`	`'c'`, `'U'`
`char` (Sonderzeichen)	`'\n'`, `'\\'`
`char` (Unicode)	`'\u1234'`
`string`	`"Dies ist ein String"`
`int`	`12`, `-128` `0xFF1F` (hexadezimal)
`long`	`12L`, `5000000000`
`float`	`12.4f`, `10e-2f`
`double`	`47.11`, `1e5`

Tabelle 3.6: Literale

Zweitens: Sie definieren eine Variable mit dem Schlüsselwort `const`:

```
const double krummeZahl = 47.11;
const double PI = 3.14159265358979323846;
```

Copyright