2 Erste Schritte

In diesem Kapitel erfahren Sie:

- ☞ Wie Sie ein Programm schreiben.
- ☞ Wie Sie ein C#-Programm übersetzen und ausführen.
- ☞ Aus welchen Elementen ein C#-Programm typischerweise aufgebaut ist.
- ☞ Welche einfachen Möglichkeiten der Programmierung mit C# existieren.

Es ist in der Welt der Programmiersprache C – und dazu zählt auch C# – Usus, mit dem ersten Beispielprogramm den Text *Hello World* am Bildschirm ausgeben zu lassen. In Anlehnung an diesen Brauch schreibt unser erstes Beispiel den Text *Hallo, viel Spaß mit C#!* auf den Bildschirm. Das ist weniger abgehoben und bringt in jedem Fall zum Ausdruck, dass Programmieren neben Arbeit auch Spaß bereiten soll.

Damit ist auch schon vorgegeben, was dieses Kapitel möchte. Es soll einen groben Einblick bieten, was die Programmiersprache C# kann, ohne hier genauer auf die einzelnen Details einzugehen. Dies bleibt den nachfolgenden Kapiteln vorbehalten. Jedoch sollte nach Bearbeitung der einzelnen Beispiele auch der Programmierneuling in der Lage sein, ähnliche Programmieraufgaben selbst zu lösen.

2.1 Hallo, viel Spaß mit C#!

Unser erstes Beispiel ist die Minimal-Version eines C#-Programms, mit dem wir den ersten Umgang mit der Sprache zeigen möchten. Das Programm kann man später sukzessive erweitern.

Quellcode: (HelloCsharp.cs)

```
1  public class HelloCsharp
2  {
3    public static void Main()
4    {
5      System.Console.WriteLine("Hallo, viel Spass mit C#!");
6    }
7  }
```

Erläuterung:

Zeile 1: Ein C#-Programm besteht aus mindestens einer Klassen-Definition. In unserem Fall wird die Klasse `HelloCsharp` definiert. Das Schlüsselwort `public` ist nicht unbedingt erforderlich. Es symbolisiert aber, dass diese Klasse von jedermann/frau aufgerufen werden kann (vgl. Abschn. 9.2.3). Bei der Programmiersprache Java ist dieses Schlüsselwort erforderlich, ebenso muss dort der Name der Klasse mit dem Dateinamen der Datei identisch sein, die die Klasse enthält. C# ist auch in dieser Hinsicht großzügiger. Wenn man sich einmal daran gewöhnt hat, wird man diese Schreibweise trotzdem beibehalten.

Zeile 2, Zeile 7: Die Klassen-Definition beginnt mit einer öffnenden geschweiften Klammer (`{`) und endet mit der zugehörigen schließenden Klammer (`}`).

Zeile 3: Die Methode, die unseren Code enthält, trägt den Namen `Main`. Dass es sich um eine Methode handelt, erkennt man an der darauf folgenden öffnenden und schließenden Klammer. Während der Name der Klasse frei wählbar war, muss die Methode, bei der die Ausführung eines C#-Programms beginnen soll, immer den Namen `Main` haben, denn der JIT-Compiler ruft zu Programmbeginn immer eine Methode namens `Main` auf. Ist keine solche Methode vorhanden, kann das Programm nicht eigenständig laufen.

Eingeleitet wird die Methode `Main` wiederum durch das Schlüsselwort `public`, dessen Verwendung wie in Zeile 1 mehr symbolisch zu verstehen ist.

Das Schlüsselwort `static` ist notwendig, da die Methode `Main` ohne ein Objekt, dem sie zugeordnet werden könnte, aufgerufen wird.

`void` wird verwendet um anzuzeigen, dass die Methode `Main` kein Ergebnis zurück gibt.

Zeile 4, Zeile 6: Die Methoden-Definition beginnt mit einer öffnenden geschweiften Klammer (`{`) und endet mit der zugehörigen schließenden Klammer (`}`).

Zeile 5: In dieser Zeile wird die Methode `WriteLine` aufgerufen, die die Zeichenkette *Hallo, viel Spaß mit C#!* am Bildschirm ausgibt. Die Methode `WriteLine` gehört zur Klasse `Console`, und diese wiederum ist im Namensraum `System` zu finden.

Die Zeichenkette wird dargestellt als Folge einzelner Zeichen, welche in Anführungszeichen (doppelte Hochkomma `"`) eingeschlossen werden.

Diese Zeile wird durch ein Semikolon abgeschlossen. Für später können Sie sich schon merken, dass alle *Anweisungen* der Sprache C# durch ein Semikolon beendet werden.

2.1.1 Editieren

Um das Programm auszuprobieren, gibt man den Text in einem Texteditor eigener Wahl ein (vgl. Abschn. 1.1.1). Dann speichert man das Programm in einer Datei namens `HelloCsharp.cs` ab. In der Regel haben C#-Programme die Dateiendung (Suffix) `.cs`, eine Vorgabe, der man folgen sollte. Die Eingabe des Programmbeispiels kann frei formatiert werden, d. h. anstelle eines Leerzeichens können beliebig viele eingegeben werden, dasselbe gilt für Leerzeilen, auch ein Zeilenumbruch ist an vielen Stellen möglich. Um die Lesbarkeit zu erhöhen, sollte man die Möglichkeit nutzen, Leerzeichen einzufügen; sie werden vom Compiler einfach ignoriert.

2.1.2 Übersetzen

Den C#-Compiler ruft man unter Windows entweder in der Visual Studio .NET Befehlszeile oder in der (DOS-)Eingabeaufforderung mit der Anweisung

```
csc HelloCsharp.cs
```

und unter Linux mit

```
mcs HelloCsharp.cs
```

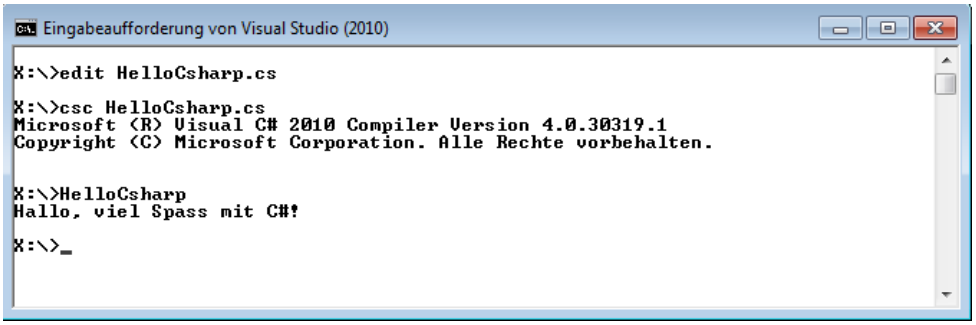
auf. Als Ergebnis erhält man eine ausführbare Datei mit Namen `HelloCsharp.exe`.

2.1.3 Ausführen

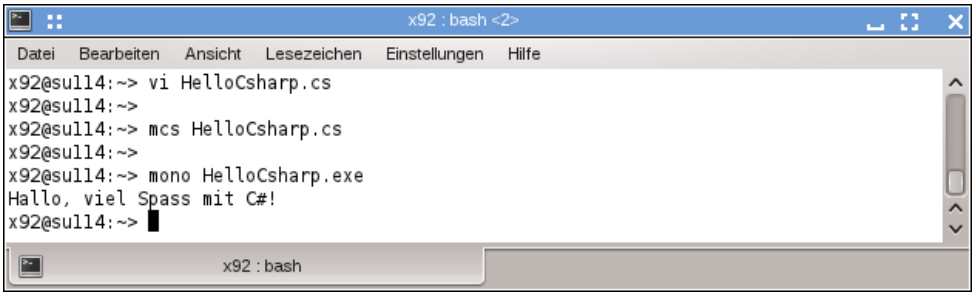
Diese Datei wird unter Windows einfach durch Aufruf ihres Namens ausgeführt, in unserem Beispiel also `HelloCsharp`. Unter Linux muss der Aufruf `mono HelloCsharp.exe` lauten. Man erhält als Ergebnis folgende Ausgabe am Bildschirm:

```
Hallo, viel Spass mit C#!
```

Zusammenfassend soll Ihnen das folgende Bild noch einmal alle Anweisungen zeigen, die Sie unter Windows eingeben müssen, um das obige Beispiel nachzuvollziehen.



Das analoge Bild für Linux:



Wie Sie aus diesem ersten Beispiel ersehen können, wird sich, wer die Programmiersprachen C, C++ oder Java kennt, bei C# fast zu Hause fühlen. Auch Umsteigern von anderen Sprachen wie Visual Basic oder Delphi wird der Wechsel zu C# nicht sonderlich schwer fallen.

2.1.4 Namensraum

Unser Programm verwendet Elemente der Klassenbibliothek des *.NET Frameworks*. Das ist die Bibliothek, die C# verwendet. Der von unserem Programm verwendete Namensraum heißt `System`. Auf Namensräume wird erst später eingegangen (siehe Abschn. 3.7 auf S. 54).

Indem wir mit der Anweisung

```
using System;
```

bestimmen, dass das Programm mit diesem Namensraum arbeitet, können wir beispielsweise den Aufruf der Methoden dieses Namensraums verkürzen.

Quellcode: (HelloCsharp0.cs)

```
1  using System;
2
3  public class HelloCsharp0
4  {
5    public static void Main()
6    {
7      Console.WriteLine("Hallo, viel Spass mit C#!");
8    }
9  }
```

Erläuterung:

Zeile 1: Durch Angabe des Namensraums `System` wird festgelegt, dass das Programm die Bezeichner bzw. Namen aus diesem Namensraum verwendet.

Zeile 7: Aus diesem Grund kann der Aufruf der Methode `WriteLine` auf die Angabe des Namens der Klasse `Console` reduziert werden.

Wir werden später sehen, dass weitere Komponenten aus anderen Namensräumen benötigt werden. Man kann sich auch in diesen Fällen durch weitere `using`-Anweisungen eine Menge Schreibarbeit ersparen.

2.2 Die C#-Eingabe

Um eine C#-Eingabe zu erstellen, muss man gewisse Regeln einhalten. Der Compiler kennt eine Anzahl von Zeichen, die er akzeptiert. Daraus werden die Schlüsselworte, Operatoren und vom Anwender definierte Worte gebildet, die später ein lauffähiges Programm ergeben.

2.2.1 Zeichenvorrat

Der Zeichenvorrat für einen Quelltext in C# umfasst:

- Alphabetische Zeichen:

  ```
  A  B  C  D  E  F  G  H  I  J  K  L  M
  N  O  P  Q  R  S  T  U  V  W  X  Y  Z
  a  b  c  d  e  f  g  h  i  j  k  l  m
  n  o  p  q  r  s  t  u  v  w  x  y  z
  _  $  @
  ```

- Numerische Zeichen:

  ```
  0  1  2  3  4  5  6  7  8  9
  ```

- Sonderzeichen:

  ```
  +  -  *  /  =  (  )  [  ]  {  }  <  >  '  "
  !  #  %  &  |  ^  ~  \  .  ,  ;  :  ?  ␣ (Leerzeichen)
  ```

Zusätzlich verwendet das *.NET Framework* zum Darstellen von Zeichen UTF-16 (Unicode Transformation Format, 16-Bit-Codierungsformular).

Der Unicode-Standard ist ein universelles Zeichenkodierungsschema. Es weist jedem Zeichen, das in den Schriftsprachen der Welt verwendet wird, einen eindeutigen numerischen Wert (Codepunkt) und einen Namen zu. Das Zeichen `A` wird beispielsweise durch den Code `u0041` und den Namen *LATIN CAPITAL LETTER A* dargestellt. Es stehen Werte für über 65 000 Zeichen zur Verfügung und es werden bis zu einer Million weitere Zeichen unterstützt. Weitere Informationen finden Sie unter *The Unicode Standard* unter der URL

```
http://www.unicode.org
```

Das Unicode-Zeichenkodierungsschema vereinfacht die Entwicklung weltweit einsetzbarer Anwendungen, da es die Darstellung aller internationalen Zeichen in einer Codierung ermöglicht. Anwendungsentwickler müssen nicht mehr das Kodierungsschema verfolgen, das für die Erstellung von Zeichen für eine bestimmte Sprache verwendet wurde. Damit können die Daten länderübergreifend auf allen Systemen verwendet werden, ohne dass beispielsweise deutsche Umlaute zu kryptischen Symbolen werden.

2.2.2 Formatierung eines C#-Programms

C# ist eine formatfreie Sprache. Es gibt fast keine Einschränkung, wie der äußere Aufbau des Programms auszusehen hat. Es liegt in der Verantwortung und auch im Interesse des Programmierers, für eine gute Lesbarkeit seiner Programme zu sorgen.

Der C#-Compiler ist ziemlich anspruchslos in Bezug auf die Übersichtlichkeit des Programmcodes. Man kann es auch so ausdrücken, dass der Compiler das Programm noch versteht, wenn der Programmierer schon lange keinen Überblick mehr hat.

Formatfrei heißt insbesondere, dass das Zeilenende nicht wie bei anderen Programmiersprachen als Anweisungsende betrachtet wird. Unser erstes Beispiel könnten wir auch wie folgt darstellen:

Quellcode: (HelloCsharpX.cs)

```
1 using System; class HelloCsharpX { static void Main() {
2 Console.WriteLine("Hallo, viel Spass mit C#!"); } }
```

Man könnte das Programm sogar in eine Zeile schreiben, nur ist diese Buchseite nicht breit genug, den Text zu fassen.

Diese Version des Programms erzeugt zwar die gleiche Ausgabe, ist aber schwerer lesbar. Man kann sich leicht vorstellen, was geschieht, wenn das Programm nicht nur aus wenigen Zeilen besteht. Es ist keine übersteigerte Ordnungsliebe oder Pedanterie, wenn man auf eine gute Lesbarkeit Wert legt. Die Erfahrung zeigt, dass man als Programmierer den Großteil seiner Zeit nicht mit Schreiben neuer Programme verbringt, sondern mit dem Weiterentwickeln und Beseitigen von Fehlern in schon erstellten Programmen. Unübersichtlich und kryptisch geschriebener Programmcode führt meist dazu, dass man ihn schon nach wenigen Wochen nur noch schwer versteht, ganz zu schweigen davon, was geschieht, wenn der Code von einem anderen Programmierer mit anderen Eigenheiten verfasst wurde.

Für gute Lesbarkeit sollten daher einige Regeln bei der Formatierung des Quellcodes unbedingt beachtet werden (eine Entwicklungsumgebung kann den Programmierer dabei unterstützen):

- Einrückungen sollten einheitlich auf eine bestimmte Anzahl von Leerzeichen festgesetzt werden. Es empfiehlt sich ein Wert von 2 bis 4 Spalten pro Einrückung.
- Programmteile sollten gemäß ihrer logischen Zusammengehörigkeit eingerückt werden.
- Die Zeilenlänge sollte 60 bis 80 Zeichen nicht überschreiten. Dies ist ein guter Wert für die gängige Bildschirmdarstellung der Editoren und gewährleistet die Lesbarkeit ohne lästiges Rechts-Links-Blättern.
- Geschweifte Klammern sollten in einer eigenen Zeile stehen, alternativ kann man aus Platzgründen die öffnende Klammer an das Ende der vorherigem Zeile ziehen. Zusammengehörige Klammern sollten des Weiteren in der gleichen Spalte stehen, damit Anfang und Ende von Programmteilen leicht zu erkennen sind; in der alternativen Version rückt man die Klammer analog ein.
- Leerzeilen und Zeilenumbrüche sollten massiv eingesetzt werden.
- Kommentare am Anfang zum Zweck des Programms und an komplizierteren Stellen im Programmcode helfen beim Verständnis des Programms auch nach längerer Zeit.

Wir wollen dies (und noch ein paar weitere Kleinigkeiten) an einigen Beispielen verdeutlichen.

Beispiel 1:

Quellcode: (HelloCsharp1.cs)

```
1  /* Abwandlung des Hallo-Welt-Programms */
2  using System;
3
4  public class HelloCsharp1
5  {
6    public static void Main()
7    {
8      Console.WriteLine("Hallo,\n"
9                        + "viel Spass mit C#!");
10   }
11 }
```

Ausgabe:

```
Hallo,
viel Spass mit C#!
```

Erläuterung:

Zeile 1: Ist ein Blockkommentar; alles zwischen `/*` und `*/` wird vom Compiler ignoriert (vgl. Abschn. 3.6)

Zeilen 8, 9: In diesen beiden Zeilen wird die Bildschirmausgabe erzeugt. Die *Escape*-Sequenz `\n` ist der Grund dafür, dass die Ausgabe in zwei Zeilen erfolgt. Die beiden Teil-Zeichenketten werden durch den Operator + (Verkettungsoperator) zu einer einzigen Zeichenkette zusammengefügt (verkettet).

Es gibt noch weitere *Escape*-Sequenzen, die zu einem späteren Zeitpunkt behandelt werden (siehe Abschn. 3.5.3 auf S. 52). An diesem Beispiel erkennt man schon, dass nicht die zweizeilige Eingabe für die zweizeilige Ausgabe verantwortlich ist, sondern einzig und allein das Einfügen von `\n`. Trotzdem sollte man aus Gründen der Übersichtlichkeit die Eingabe so formatieren, dass schon eine Vorstellung der zu erwartenden Ausgabe sichtbar wird.

Beispiel 2:

Quellcode: (HelloCsharp2.cs)

```
1  /* Weitere Abwandlung des Hallo-Welt-Programms */
2  using System;
3
4  public class HelloCsharp2
5  {
6    public static void Main()
7    {
8      Console.WriteLine("******************************\n"
9                        + "*                            *\n"
10                       + "*  Hallo, viel Spass mit C#!  *\n"
11                       + "*                            *\n"
12                       + "******************************");
13   }
14 }
```

Ausgabe:

```
******************************
*                            *
*  Hallo, viel Spass mit C#!  *
*                            *
******************************
```

Erläuterung:

Zeilen 8–12: In diesen Zeilen wird wiederum die Bildschirmausgabe erzeugt. Auch hier ist die *Escape*-Sequenz `\n` der Grund, dass die Ausgabe in mehreren Zeilen erfolgt.

In Beispiel 2 gibt die Methode `WriteLine` eine Zeichenkette aus, die sich in der Eingabe über fünf Zeilen erstreckt. Jede Zeile enthält auch hier wieder eine Teil-Zeichenkette. Die Leerzeichen innerhalb der Zeichenkette sind signifikant, d. h. sie werden so ausgegeben, wie sie in der Zeichenkette stehen.

Beispiel 3:

Quellcode: (HelloCsharp3.cs)

```
1  /* Mehrfacher Aufruf einer Methode (unnoetig) */
2  using System;
3
4  public class HelloCsharp3
5  {
6    public static void Main()
7    {
8      Console.WriteLine("******************************");
9      Console.WriteLine("*                            *");
10     Console.WriteLine("*  Hallo, viel Spass mit C#!  *");
11     Console.WriteLine("*                            *");
12     Console.WriteLine("******************************");
13   }
14 }
```

Ausgabe:

```
******************************
*                            *
*  Hallo, viel Spass mit C#!  *
*                            *
******************************
```

Erläuterung:

Zeilen 8–12: In diesen Zeilen wird ebenso wie in Beispiel 2 die Bildschirmausgabe erzeugt. Hier ist allerdings der wiederholte Aufruf der Methode `WriteLine` der Grund, dass die Ausgabe in mehreren Zeilen erfolgt.

Wie man sieht, liefern Beispiel 2 und Beispiel 3 die gleiche Ausgabe. Der Unterschied liegt im Erzeugen der Ausgabe. In Beispiel 2 wird einmal die Methode `WriteLine` aufgerufen, im Beispiel 3 dagegen wird die Methode `WriteLine` fünfmal verwendet. Damit stellt sich die Frage: Was ist besser? Es ist sicherlich richtig, dass der fünfmalige Aufruf von `WriteLine` ein größerer Aufwand für den Computer ist. Wenn man diesen Aufwand allerdings mit dem Gesamtaufwand für eine Bildschirmausgabe vergleicht, ist dieser Aufwand minimal. Trotzdem ist im Interesse einer sauberen Programmierung eine Kodierung wie in Beispiel 2 vorzuziehen.

2.3 Rechnen mit C#

C# kann jedoch viel mehr, als lediglich Texte am Bildschirm auszugeben. Wie jede andere Programmiersprache kann man mit C# rechnen – sowohl mit ganzen als auch mit Gleitkommazahlen, wie die beiden folgenden Beispiele zeigen.

2.3.1 Wieviel ist 22 plus 38?

Das folgende Beispiel zeigt die Verwendung von ganzen Zahlen. Diese werden in Variablen im Speicher abgelegt.

Quellcode: (Addition.cs)

```
1  /* Addition zweier ganzer Zahlen */
2  using System;
3
4  public class Addition
5  {
6    public static void Main()
7    {
8      int zahl1, zahl2, summe;
9
10     zahl1=22;
11     zahl2=38;
12
13     summe=zahl1+zahl2;
14
15     Console.WriteLine(zahl1 + "+" + zahl2 + "=" + summe);
16   }
17 }
```

Ausgabe:

```
22+38=60
```

Erläuterung:

Zeile 8: Durch die Anweisung `int`, werden im *Stack* drei Speicherplätze für drei ganze Zahlen reserviert. Mit den Namen `zahl1`, `zahl2` und `summe` wird auf diese Speicherplätze zugegriffen.

Zeilen 10, 11: In der Variablen `zahl1` wird die Zahl 22, in der Variablen `zahl2` die Zahl 38 gespeichert.

Zeile 13: Der Inhalt der Variablen `zahl1` und `zahl2` wird addiert und das Ergebnis in der Variablen `summe` abgelegt.

Zeile 15: Der Inhalt der Variablen `zahl1`, das Zeichen Plus, der Inhalt der Variablen `zahl2`, das Gleichheitszeichen und der Inhalt der Variablen `summe` werden verkettet (zu einer Zeichenkette zusammengefügt) und mit der Methode `WriteLine` am Bildschirm ausgegeben.

Die Zuweisung der Werte an die Variablen geschieht mit dem Zuweisungsoperator (=). Mit ihm wird der Wert, der auf der rechten Seite des Gleichheitszeichens steht, in der Variablen, die links steht, abgespeichert. Stimmen die beiden Seiten in ihrem Datentyp nicht überein, so erfolgt, falls möglich, eine Datentypkonvertierung. Ist diese Datentypkonvertierung nicht möglich, wird das Programm mit einer Fehlermeldung beendet.

Beim Aufbau der Zeichenkette werden die Variablen `zahl1`, `zahl2` und `summe` in den Datentyp einer Zeichenkette konvertiert, damit sie mit den anderen Teilen verkettet werden können.

Nun dasselbe Beispiel wie oben, allerdings ohne den Einsatz von Variablen:

Quellcode: (Ausgabe01.cs)

```
1  /* Rechnen in der Ausgabe */
2  using System;
3
4  public class Ausgabe01
5  {
6    public static void Main()
7    {
8      Console.WriteLine("Die Summe von 22 und 38 ist " + 22+38 + ".");
9    }
10 }
```

Ausgabe:

```
Die Summe von 22 und 38 ist 2238.
```

Erläuterung:

Zeile 8: Hier soll die Summe von 22 und 38 direkt berechnet und ausgegeben werden. Die Ausgabe zeigt, dass dies nicht geschieht.

Sowohl die Zahl 22 als auch die Zahl 38 werden in eine Zeichenkette konvertiert und mit dem Rest der Zeichenkette ausgegeben. Eine Addition findet daher nicht statt, da das Plus-Zeichen als Verkettungsoperator interpretiert wurde.

Um zu verhindern, dass die beiden Zahlen 22 und 38 zuerst in eine Zeichenkette konvertiert werden und danach zu 2238 verkettet werden, muss man dafür sorgen, dass die Addition zuerst durchgeführt wird.

Quellcode: (Ausgabe02.cs)

```
7    {
8      Console.WriteLine("Die Summe von 22 und 38 ist " + (22+38) + ".");
9    }
```

Ausgabe:

```
Die Summe von 22 und 38 ist 60.
```

Erläuterung:

Zeile 8: Hier ist der Ausdruck `22+38` in runde Klammern eingeschlossen.

Durch die Klammerung wird erreicht, dass zuerst addiert wird. Die Ausgabe bzw. die Konvertierung zur Ausgabe erfolgt erst danach.

Bei diesem Beispiel ist zum ersten Mal in dieser Einführung nicht der gesamte Quellcode des Programms abgedruckt. Dies ist zum Verständnis nicht nötig und würde nur Platz verbrauchen. Wir werden künftig häufig so verfahren, wenn der Programmteil allein genügend Information vermittelt. Wenn Sie die Beispielprogramme abtippen und ausprobieren möchten, müssen Sie die fehlenden Zeilen ergänzen. Sie finden meist ein paar Seiten vorher ein vollständiges Beispiel, so dass es recht einfach ist, das vollständige Programm zu erstellen. Alternativ können Sie sich die vollständigen Beispiele auch aus dem Netz herunterladen (näheres im Vorwort, S. i).

2.3.2 Wie weit war es nach Marathon?

Auch das folgende Beispiel wäre, ebenso wie das vorherige, schneller im Kopf zu rechnen oder mit einem Taschenrechner zu bearbeiten. Man kann aber hieran das Rechnen mit Gleitkommazahlen nachvollziehbar darstellen. Es geht bei der Aufgabe darum (ohne auf die geschichtlichen Hintergründe einzugehen), die Länge eines Marathonlaufs, die mit 26 Meilen und 285 Yards angegeben wird, in das bei uns übliche Maßsystem umzurechnen, nämlich in Kilometer. Die Formel zur Umrechnung lautet

$$\text{Kilometer} = 1,609 \times (\text{Meilen} + \text{Yards}/1760).$$

Quellcode: (Marathon.cs)

```
1  /* Umrechnung der Marathonlaufstrecke */
2  using System;
3
4  public class Marathon
5  {
6    public static void Main()
7    {
8      double meilen, yards, kilometer;
9
10     meilen=26.0;
11     yards=385.0;
12
13     kilometer=1.609*(meilen+yards/1760.0);
14
15     Console.WriteLine("Die Länge des Marathonlaufs beträgt "
16                       + kilometer + " km.");
17   }
18 }
19
```

Ausgabe:

```
Die Länge des Marathonlaufs beträgt 42.18596875 km.
```

Erläuterung:

Zeile 8: Mit der Anweisung `double` wird unter den Namen `meilen`, `yards` und `kilometer` im Stack für drei Variablen Speicherplatz für Gleitkommazahlen reserviert. Gleitkommazahlen sind für uns vorerst Zahlen, die ein Komma und Nachkommastellen enthalten. Sie werden oft auch als Gleitpunktzahlen bezeichnet, da im englischen Sprachraum statt des Kommas ein Punkt gesetzt wird. Sie können das auch im Programm sehen, auch hier wird ein Punkt geschrieben.

Zeilen 10, 11: Die Variablen `meilen` und `yards` werden initialisiert. Es sollte Ihnen auffallen, dass die Zahlen 26 und 385 als 26.0 und 385.0, d. h. ebenfalls als Gleitkommazahlen angegeben werden.

Zeile 13: In dieser Zeile wird die Umrechnung nach obiger Formel vorgenommen. Auch hier wird die Zahl 1760 als Gleitkommazahl, d. h. 1760.0 geschrieben.

Zeilen 15, 16: Mit der Methode `WriteLine` wird, wie schon erklärt, die Ausgabe erzeugt, wobei der Inhalt der Variablen `kilometer` wieder in eine Zeichenkette konvertiert wird und dann mit der Zeichenkette davor und danach verkettet wird.

Gleitkommazahlen und ganze Zahlen werden im Rechner in einem unterschiedlichen Format gespeichert. Daher muss beim Wechsel von der einen in die andere Form eine Datentypkonvertierung vorgenommen werden. Die Konvertierung von ganzzahligen Zahlen nach Gleitkommazahlen wird automatisch vorgenommen – kostet allerdings Rechenzeit. Der routinierte Programmierer vermeidet (wenn möglich) solche Fälle. Bei der Zuweisung der Werte für die Variablen `meilen` und `yards` ist es daher sinnvoll, statt einem ganzzahligen Literal (vgl. Abschn. 3.5) eine Gleitkommazahl zu verwenden.

2.4 Eingabe von Daten

Als Vorgriff auf zukünftige Sprachelemente und ohne weitere Erklärung an dieser Stelle (sie folgt später in diesem Handbuch) sind die folgenden Beispiele von Nutzen. Wenn man programmiert, möchte man Daten an die Programme übergeben. Hierbei ist zu unterscheiden, ob die Daten beim Aufruf des Programms mit übergeben oder während des Programmlaufs über die Tastatur eingegeben werden.

2.4.1 Eingabe beim Programmaufruf

Bei einer Parameterübergabe beim Programmaufruf werden die Werte an die Methode `Main()` übergeben. Aus diesem Grund wird der Aufruf der Methode `Main` von

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

zu

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

geändert. Dadurch wird ein Array `args` (args = Kurzform für *arguments*) übergeben, dem die Eingabewerte zugeordnet und von der Methode `Main` ausgelesen werden können. Anstelle des Arraynamens `args` können auch andere Namen verwendet werden, doch hat sich dieser Name eingebürgert.

Quellcode: (Parameter.cs)

```
1  /* Parameteruebergabe an ein Programm */
2  using System;
3
4  public class Parameter
5  {
6    public static void Main(string [] args)
7    {
8      Console.WriteLine("Anzahl der Parameter: " + args.Length + "\n"
9                        + "Erster Parameter:     " + args[0]     + "\n"
10                       + "Zweiter Parameter:    " + args[1]     + "\n"
11                       + "Dritter Parameter:    " + args[2]);
12   }
13 }
14
```

Die einzelnen Parameter, die man beim Programmaufruf übergibt, werden durch Leerzeichen getrennt.

Aufruf:

```
Parameter eins zwei drei
```

Ausgabe:

```
Anzahl der Parameter: 3
Erster Parameter:     eins
Zweiter Parameter:    zwei
Dritter Parameter:    drei
```

Erläuterung:

Zeilen 8–11: Die Parameter, die beim Aufruf übergeben wurden, werden ausgewertet. Der für die Übergabe verwendete Arrayname ist `args`. Mit `args.Length` erhält man die Anzahl der übergebenen Parameter, in diesem Fall sind das drei. Die einzelnen Parameter bekommt man mit `args[0]` für den ersten, `args[1]` für den zweiten und `args[2]` für den dritten Parameter.

Hier ist festzuhalten, dass in C# immer Zeichenketten übergeben werden. Auch beim Aufruf von

```
ParameterI 11111 22222
```

sind die übergebenen Werte keine Zahlen, sondern Zeichenketten. Um mit diesen Werten rechnen zu können, müssen sie zuerst in Datentypen für Zahlen konvertiert werden.

Quellcode: (ParameterI.cs)

```
1  /* Parameteruebergabe und Summe von ganzen Zahlen */
2  using System;
3
4  public class ParameterI
5  {
6    public static void Main(string [] args)
7    {
8      int zahl1, zahl2;
9
10     zahl1 = Int32.Parse(args[0]);
11     zahl2 = Int32.Parse(args[1]);
12
13     Console.WriteLine("Die Summe der Zahlen ist: " + (zahl1 + zahl2));
14   }
15 }
```

Aufruf:

```
ParameterI 11111 22222
```

Ausgabe:

```
Die Summe der Zahlen ist: 33333
```

Erläuterung:

Zeile 8: Es wird Speicherplatz für zwei ganze Zahlen unter dem Namen `zahl1` und `zahl2` reserviert.

Zeile 10: Der Wert, der als erster Parameter übergeben wurde, d. h. der in `args[0]` enthalten ist, wird durch die Methode `Int32.Parse()` in eine ganze Zahl konvertiert und dann der Variablen `zahl1` zugewiesen.

Zeile 11: Das Gleiche geschieht mit dem zweiten Parameter, das Ergebnis wird der Variablen `zahl2` zugewiesen.

Zeile 13: Es wird die Summe der Variablen `zahl1` und `zahl2` berechnet und das Ergebnis am Bildschirm ausgegeben.

Zusammenfassend ist festzuhalten: Beim Konvertieren muss der Datentyp der Variablen berücksichtigt werden.

Im nächsten Beispiel wird der Datentyp `double` statt `int` verwendet.

Quellcode: (ParameterD.cs)

```
8       double zahl1, zahl2;
9
10      zahl1 = Double.Parse(args[0]);
11      zahl2 = Double.Parse(args[1]);
12
13      Console.WriteLine("Die Summe der Zahlen ist: " + (zahl1 + zahl2));
```

Aufruf:

```
ParameterD 1.1111 2.2222
```

Ausgabe:

```
Die Summe der Zahlen ist: 3.3333
```

Erläuterung:

Zeile 8: Es werden zwei Variablen mit dem Datentyp `double` angelegt.

Zeilen 10, 11: Mit der Methode `Double.Parse()` werden die Parameter `args[0]` und `args[1]` in den Datentyp `double` konvertiert.

2.4.2 Eingabe über die Tastatur

Möchte man Daten während des Programmlaufs über die Tastatur einlesen, setzt man die Methode `Console.ReadLine()` ein. Auch in diesem Fall wird wieder eine Zeichenkette übergeben. Bevor man eine Zeichenkette einliest, sollte man auf dem Bildschirm einen entsprechenden Text ausgeben, nämlich dass das Programm jetzt eine Eingabe erwartet. Sonst könnte es passieren, dass zwei warten: Der Rechner, d. h. das Programm auf eine Eingabe, und der Anwender darauf, dass es weiter geht.

Quellcode: (Tastatur.cs)

```
1  /* Eingabe ueber die Tastatur */
2  using System;
3
4  public class Tastatur
5  {
6    public static void Main()
7    {
8      Console.WriteLine("Bitte Zeichenkette eingeben:");
9      Console.WriteLine("Ausgabe: " + Console.ReadLine());
10   }
11 }
12
```

Ein- und Ausgabedialog:

```
Bitte Zeichenkette eingeben:
```

Dann erfolgt die Eingabe

```
Das ist eine Zeichenkette.
```

Ausgabe:

```
Ausgabe: Das ist eine Zeichenkette.
```

Erläuterung:

Zeile 8: Es wird am Bildschirm ausgegeben, dass das Programm eine Eingabe erwartet.

Zeile 9: Die Eingabe wird durch die Methode `Console.ReadLine()` von der Tastatur eingelesen und mit dem vorangestellen Text, der Zeichenkette „Ausgabe“, am Bildschirm ausgegeben.

Auch für die Eingabe von der Tastatur gilt, dass die eingegebenen Werte Zeichenketten sind. Also müssen auch hier eingegebene Werte vor der Benutzung in die entsprechende Zahl konvertiert werden.

Quellcode: (TastaturI.cs)

```
8       int zahl1, zahl2;
9
10      Console.WriteLine("Bitte erste Zahl eingeben:");
11      zahl1 = Int32.Parse(Console.ReadLine());
12      Console.WriteLine("Bitte zweite Zahl eingeben:");
13      zahl2 = Int32.Parse(Console.ReadLine());
14
15      Console.WriteLine("Die Summe der Zahlen ist: " + (zahl1 + zahl2));
```

Ein- und Ausgabedialog:

```
Bitte erste Zahl eingeben:
11111
Bitte zweite Zahl eingeben:
22222
```

Ausgabe:

```
Die Summe der Zahlen ist: 33333
```

Erläuterung:

Zeile 8: Es wird Speicherplatz für zwei ganze Zahlen reserviert.

Zeilen 10–13: Es werden zwei Zahlen als Zeichenketten von der Tastatur eingelesen, konvertiert und den Variablen zugewiesen.

Zeile 15: Es wird die Summe gebildet und am Bildschirm ausgegeben.

2.5 Eine einfache Entscheidung

In allen Programmen, die bisher bearbeitet wurden, sind die einzelnen Anweisungen (genauer Anweisungszeilen) nacheinander abgearbeitet worden. Doch ist es manchmal sinnvoll, wie das folgende Programm zeigt, von dieser sequentiellen Abarbeitungsweise abzuweichen.

Im folgenden Programm werden zwei Zahlen übergeben und die erste durch die zweite Zahl geteilt. Daher ist es wichtig dafür zu sorgen, dass die zweite Zahl ungleich Null ist.

Quellcode: (Verzweig.cs)

```
1  /* Bedingte Programmausführung */
2  using System;
3
4  public class Verzweig
5  {
6    public static void Main(string [] args)
7    {
8      int zahl1, zahl2;
9
10     zahl1 = Int32.Parse(args[0]);
11     zahl2 = Int32.Parse(args[1]);
12
13     if(zahl2 != 0)
14     {
15       Console.WriteLine(zahl1 + "/" + zahl2 + "=" + zahl1/zahl2);
16     }
17   }
18 }
```

Aufruf:

```
Verzweig 6 2
```

Ausgabe:

```
6/2=3
```

Erläuterung:

Zeile 13–16: Mit der Anweisung `if` wird geprüft, ob die Bedingung `zahl2!=0` (in Worten `zahl2` ungleich Null) erfüllt ist. Wenn das zutrifft, wird der nachfolgende Block (geschweifte Klammern) ausgeführt, der nur eine Anweisung enthält. Diese Anweisung führt die Division durch und gibt das Ergebnis am Bildschirm aus.

Mit Hilfe der Anweisung `if` ist der Programmierer in der Lage, die sequentielle Abarbeitung zu unterbrechen, also eine Verzweigung in das Programm einzubringen. Man hat die Möglichkeit, eine Anweisung nur unter gewissen Voraussetzungen auszuführen.

- Gesteuert wird die Anweisung `if` durch die Bedingung, die auf `if` folgt, sie wird in runde Klammern eingeschlossen.
- Ist die Bedingung `wahr`, wird die nachfolgende Anweisung ausgeführt, ist sie `falsch`, wird sie übergangen.
- Soll mehr als eine Anweisung in Abhängigkeit von `if` ausgeführt werden, so muss man diese Anweisungen in geschweifte Klammern einschließen. In diesem Beispiel könnte man also die Klammern auch weglassen. Es erhöht aber deutlich die Lesbarkeit, unabhängig von der Anzahl der Anweisungen immer die Klammern zu setzen.

2.6 Wieviel ist die Summe aller Zahlen von 1 bis 100?

Als der berühmte Mathematiker Carl Friedrich Gauss im Alter von zehn Jahren in der Schule die Zahlen von 1 bis 100 addieren sollte, hatte er innerhalb kürzester Zeit die Lösung auf seine Tafel geschrieben: 5050. Seine Mitschüler brauchten sehr lange für ihre Berechnungen, hatten ziemlich verschmierte Tafeln und keiner hatte das richtige Ergebnis herausbekommen. Gauss hatte erkannt, so vermutet man, dass jedes der Zahlenpaare 1 und 100, 2 und 99, 3 und 98, 4 und 97,

usw. bis zu 50 und 51, zusammen 101 ergeben. Daher ist die Gesamtsumme der 50 Paare 50·101 = 5050.

Wenn man ein Programm schreibt, um Zahlen zu addieren, so wird ein Computer dies so tun, wie es der Programmierer ihm vorgibt, d. h. ganz stur alle Zahlen einzeln addieren. Da Computer aber sehr schnell sind, bekommt man innerhalb kürzester Zeit die Lösung, egal wieviele Zahlen zu addieren sind.

Zunächst soll unser Programm eine viel kleinere Zahl von Werten aufsummieren: Die Zahlen von 1 bis 5. Um diese Summe zu bestimmen, muss man folgende Rechnung durchführen:

```
1 + 2 + 3 + 4 + 5
```

Wie man leicht im Kopf ausrechnen kann, ist die Summe 15.

Um mit dem Computer eine solche Rechnung durchzuführen, kann man eine Schleife verwenden. Mit einer Schleife kann man Programmteile wiederholt ausführen lassen. Dazu soll die `for`-Schleife benutzt werden. Die einfachste Form der for-Schleife, die auch hier Verwendung findet, ist

```
for (Initialisierung; Bedingung; Wiederholung)
  Schleifenanweisung
```

- Die *Initialisierung* setzt die Schleifenvariable auf einen Startwert;
- die *Bedingung* entscheidet, ob die Schleife ausgeführt werden soll und
- die *Wiederholung* gibt an, wie die Schleifenvariable bei jedem Schleifendurchgang verändert werden soll.

Summe von 1 bis 5

Quellcode: (Sum2five.cs)

```
1  /* Summe der ganzen Zahlen von 1 bis 5 */
2  using System;
3
4  public class Sum2five
5  {
6    public static void Main()
7    {
8      int zahl,summe;
9
10     summe=0;
11
12     for(zahl=1; zahl<=5; zahl=zahl+1)
13     {
14       summe = summe+zahl;
15     }
16
17     Console.WriteLine("Die Summe der Zahlen ist " + summe + ".");
18   }
19 }
```

Ausgabe:

```
Die Summe der Zahlen ist 15.
```

Erläuterung:

Zeile 8: Es wird Speicherplatz für die Variablen `zahl` (diese Variable soll die sogenannte Schleifenvariable sein) und `summe` (in dieser Variable sollen die Zahlen von 1 bis 5 aufsummiert werden) reserviert.

Zeile 10: Die Variable `summe` erhält den Startwert 0 zugewiesen.

Zeile 12: Durch die `for`-Anweisung wird die Schleife gesteuert. Die Schleifenvariable `zahl` erhält den Startwert 1. In der Bedingung wird überprüft, ob der Inhalt der Schleifenvariablen kleiner oder gleich fünf ist. Wenn das zutrifft, wird die Anweisung ausgeführt. Am Ende der Ausführung der Schleifenanweisung wird die Schleifenvariable um 1 erhöht.

Zeile 13, Zeile 15: Diese Zeilen bilden den Körper der Schleife. Wie in der Anweisung `if` wird auch bei der `for`-Schleife eine Anweisung in Abhängigkeit von einer Bedingung ausgeführt. Soll mehr als eine Anweisung in Abhängigkeit von `for` ausgeführt werden, so muss man diese Anweisungen in geschweifte Klammern einschließen. Oft sieht man auch eine einzelne Schleifenanweisung schon in geschweiften Klammern eingeschlossen, das ist zwar unnötig, kann aber gerade für den Anfänger die Zugehörigkeit zur Schleife optisch verdeutlichen.

Zeile 14: Hier werden nun die einzelnen Zahlen aufsummiert. Beim ersten Schleifendurchgang hat `zahl` den Wert 1, der zur Summe addiert wird. Beim zweiten Durchgang hat `zahl` den Wert 2, usw.

Zeile 17: Das Ergebnis wird ausgegeben.

Am Beginn jedes Schleifendurchlaufs wird geprüft, ob die Bedingung – in unserem Fall `zahl<=5` – erfüllt ist. Falls das Ergebnis dieser Prüfung wahr ist, werden die Schleifenanweisungen ausgeführt. Ist das Ergebnis falsch, wird die Schleife beendet, und es wird bei der auf die Schleife folgenden Anweisung fortgefahren. In diesem Beispiel ist das der Fall, wenn der Wert der Variablen `zahl` 6 ist.

Wenn wir in der Lage sind, die Zahlen von 1 bis 5 in einer Schleife aufzusummieren, ist es nicht schwierig, die Schulaufgabe des kleinen Gauss, wenn auch nicht so genial, doch auch bestimmt genau so schnell zu lösen.

Summe von 1 bis 100

Quellcode: (Sum2hund.cs)

```
1  /* Summe der ganzen Zahlen von 1 bis 100 */
2  using System;
3
4  public class Sum2hund
5  {
6    public static void Main()
7    {
8      int zahl,summe;
9
10     summe=0;
11
12     for(zahl=1; zahl<=100; zahl=zahl+1)
13     {
14       summe = summe+zahl;
15     }
16
17     Console.WriteLine("Die Summe der Zahlen ist " + summe + ".");
18   }
19 }
```

Ausgabe:

```
Die Summe der Zahlen ist 5050.
```

Erläuterung:

Zeile 12: Diese Zeile mit der `for`-Anweisung steuert die Schleife. Es ist lediglich notwendig, die Bedingung auf `zahl <= 100` abzuändern. Dadurch wird die Schleifen-Bedingung für alle Zahlen von 1 bis 100 wahr, für 101 wird sie falsch.

Sollen nun statt der Zahlen von 1 bis 100 beispielsweise die Zahlen von 1 bis 100 000 000 aufsummiert werden, wäre dies nun auch kein Problem mehr. Dazu müsste lediglich die Bedingung auf `zahl <= 100000000` geändert werden.

Möchte man die Zahlen von 100 000 000 bis 200 000 000 addieren, dann müsste die `for`-Anweisung lauten:

```
for (zahl=100000000; zahl<=200000000; zahl=zahl +1)
```

2.7 Unser erstes Fenster

Auch wenn das nun folgende Beispiel den Rahmen dieses Buches übersteigt, gehört es zum Rundgang durch C#, den Ihnen dieses Kapitel bieten sollte. Es soll einen Ausblick geben, welche Möglichkeiten die Programmiersprache C# bietet.

Es ist ein einfaches Beispiel, das die Ausgabe `Hallo, viel Spaß mit C#!` in einem eigenen Fenster und dazu noch bunt am Bildschirm darstellt. In diesem Buch kann es natürlich nur in Schwarz-Weiß erscheinen.

Eine Erläuterung des Quellcodes würde an dieser Stelle zu weit führen.

Quellcode: (HelloCsharp9.cs)

```
1  /* Komplexes Beispiel einer Fensterausgabe mit Windows Forms */
2  using System;
3  using System.Windows.Forms;
4  using System.Drawing;
5
6  public class HelloCsharp9 : Form
7  {
8    public static void Main()
9    {
10     Application.Run(new HelloCsharp9());
11   }
12
13   public HelloCsharp9()
14   {
15     this.Text = "Hallo, viel Spass mit C#!";
16   }
17
18   protected override
19     void OnPaint(PaintEventArgs p)
20   {
21     p.Graphics.FillRectangle(
22       new SolidBrush(Color.Blue),this.ClientRectangle);
23     Font f = new Font("Helvetica",24.0F,GraphicsUnit.Point);
24     p.Graphics.DrawString("Hallo,",f,
25       new SolidBrush(Color.Yellow),20.0F,40.0F);
26     p.Graphics.DrawString("viel Spass",f,
```

```
27          new SolidBrush(Color.Red),70.0F,100.0F);
28        p.Graphics.DrawString("mit C#!",f,
29          new SolidBrush(Color.Cyan),160.0F,160.0F);
30      }
31  }
```

2.8 Übungsaufgaben

1. Schreiben Sie das Beispielprogramm zur Berechnung der *Länge eines Marathonlaufs in Kilometern* ab.

   Rufen Sie den C#-Compiler auf, und sehen Sie sich das Ergebnis Ihres Programms auf dem Bildschirm an. Sollten Sie nicht das erwartete Ergebnis erhalten, so versuchen Sie, den Fehler zu korrigieren.

   War alles auf Anhieb fehlerfrei, so versehen Sie es absichtlich mit einem Syntaxfehler und compilieren Sie das Programm noch einmal. „Vergessen“ Sie beispielsweise ein Semikolon. Beachten Sie die Fehlermeldung(en) des Compilers. Der Compiler sagt Ihnen auch genau, in welcher Zeile der Fehler aufgetreten ist.

2. Berechnen Sie den Durchschnitt der folgenden Zahlen

   $$5, \quad 9, \quad 4, \quad 17, \quad 3,$$

   die Sie den Variablen a, b, c, d und e zugewiesen haben.
   (Ergebnis: $7,6$)

3. Gegeben sind die ganzen Zahlen 1234 und 5678. Berechnen Sie Summe, Differenz, Produkt und Quotienten dieser Zahlen und geben Sie die Ergebnisse am Bildschirm aus.

4. Gegeben sind die Gleitkommazahlen 1,234 und 5,678. Berechnen Sie Summe, Differenz, Produkt und Quotienten dieser Zahlen und geben Sie die Ergebnisse am Bildschirm aus.

5. Schreiben Sie ein Programm, das nach der Eingabe der gefahrenen Kilometer und der Menge des verbrauchten Kraftstoffs den Durchschnittsverbrauch auf 100 km berechnet und ausgibt.