C# ISBN 3-8273-1856-4

3 Grundlagen der Sprache

Nachdem Sie in Kapitel 2 erfahren haben, wie Sie C#-Programme grundsätzlich aufbauen und kompilieren, beschreibe ich in diesem Kapitel die Grundlagen der Sprache C#. Ich zeige, wie Sie Anweisungen schreiben, mit Datentypen umgehen, mit Schleifen und Verzweigungen arbeiten und Methoden (Funktionen) schreiben. Um die Beispiele möglichst einfach zu halten, verwende ich dazu einfache Konsolenanwendungen.

Ich gehe in diesem Kapitel davon aus, dass Sie die Grundlagen der Programmierung beherrschen. Lesen Sie u. U. mein Buch »Programmieren« aus der Nitty-Gritty-Reihe, wenn Sie einzelne der hier beschriebenen Themen nicht verstehen oder die Grundlagen vertiefen wollen.

Methoden (Funktionen) werden übrigens noch nicht in diesem Kapitel, sondern erst bei der objektorientierten Programmierung im nächsten Kapitel beschrieben

3.1 Umgang mit Assemblierungen und Namensräumen

Wenn Sie die in einer Assemblierung enthaltenen Typen im Programm verwenden wollen, müssen Sie diese Assemblierung referenzieren. Die einzige Assemblierung, die nicht referenziert werden muss, weil der Compiler das automatisch macht, ist *mscorlib.dll*. Diese Assemblierung enthält die grundlegenden Typen des .NET-Frameworks in verschiedenen Namensräumen. In Visual Studio verwenden Sie zur Referenzierung anderer Assemblierungen den `Referenzen`-Eintrag im Projektmappen-Explorer, so wie ich es bereits in Kapitel 2 gezeigt habe. Wenn Sie den Kommandozeilencompiler verwenden, geben Sie die zu referenzierenden Assemblierungen im reference-Argument an.

Wenn Sie die in einem Namensraum enthaltenen Typen in Ihrem Programm verwenden wollen, können Sie diese unter Angabe des Namensraums voll qualifiziert angeben. Wenn Sie beispielsweise über die `Show`-Methode der `MessageBox`-Klasse eine Meldung ausgeben wollen, können Sie die entsprechende Anweisung so schreiben:

```
System.Windows.Forms.MessageBox.Show("Hello World");
```

Sie können den Namensraum aber auch über eine `using`-Direktive, die ganz oben im Quellcode stehen muss, quasi importieren:

```
using System.Windows.Forms;
```

Dann können Sie alle darin enthaltenen Typen ohne Angabe des Namensraums verwenden:

```
MessageBox.Show("Hello World");
```

Falls der Compiler dann eine Kollision mit einem Typnamen eines anderen Namensraums meldet, können Sie diesen Typ immer noch voll qualifiziert angeben.

Mit der `using`-Direktive können Sie für einen Namensraum oder eine Klasse auch noch einen Alias einrichten:

```
using MB = System.Windows.Forms.MessageBox;
```

Den Namensraum bzw. die Klasse können Sie dann über den Alias referenzieren:

```
MB.Show("Hello World");
```

Ich verwende diese Technik nicht, weil es dann schwer zu erkennen ist, welche Klasse bzw. welcher Namensraum tatsächlich verwendet wird.

Der Windows Class Viewer

Der Windows Class Viewer ist ein Tool, über das Sie die Namensräume des .NET-Framework nach Klassen mit einem bestimmten Namen durchsuchen können. Wenn Sie einmal nicht wissen, in welchem Namensraum eine Klasse verwaltet wird, suchen Sie einfach danach. Der Class Viewer zeigt nicht nur den Namensraum, sondern gleich noch die komplette Deklaration der Klasse an. Sie finden dieses Tool unter dem Namen *WinCv.exe* im Ordner *\Programme\Microsoft Visual Studio .NET\FrameworkSDK\Bin*.

3.2 Anweisungen, Ausdrücke und Operatoren

3.2.1 Anweisungen

Elementare Anweisungen

Die Mutter aller Programme, die elementare Anweisung, entspricht syntaktisch der Anweisung von C++ oder Java. Eine C#-Anweisung wird immer mit einem Semikolon abgeschlossen:

```
Console.WriteLine("Hello World");
```

Daraus folgt, dass Sie Anweisungen einfach in die nächste Zeile umbrechen können:

```
Console.WriteLine(
   "Hello World");
```

Das Umbrechen in einer Zeichenkette ist allerdings nicht ganz so einfach. Die folgende Anweisung:

```
Console.WriteLine("Hello
   World");
```

ergibt direkt mehrere Syntaxfehler. U. a. meldet der Compiler (bzw. Visual Studio) den Fehler, dass ein »(« erwartet wird.

Zeichenketten müssen prinzipiell immer in der aktuellen Zeile abgeschlossen und in einer neuen Zeile wieder begonnen werden (außer, Sie verwenden so genannte wortwörtliche Zeichenketten, wie ich es auf Seite 110 beschreibe). Wollen Sie eine Zeichenkette umbrechen (was in der Praxis sehr häufig vorkommt), schließen Sie diese ab und verbinden sie mit der Zeichenkette in der nächsten Zeile über ein +. Die folgende Anweisung ist korrekt:

```
Console.WriteLine("Hello " +
   "World");
```

Elementare Anweisungen sind, wie Sie ja wahrscheinlich bereits wissen, entweder Zuweisungen von arithmetischen oder logischen Ausdrücken an Variablen:

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

oder Aufrufe von Funktionen/Methoden:

```
Console.WriteLine("Hello World");
```

Arithmetische und logische Ausdrücke arbeiten mit den ab Seite 87 beschriebenen Operatoren. Das Beispiel verwendet die Operatoren = (Zuweisung) und + (Addition). Den Aufruf von Methoden beschreibe ich ab Seite 85.

Daneben existieren noch die Struktur-Anweisungen (Schleifen und Verzweigungen), die ich ab Seite 143 beschreibe.

Bei allen Anweisungen müssen Sie beachten, dass C# Groß- und Kleinschreibung unterscheidet. Jedes Element einer Anweisung muss genau so geschrieben werden, wie es ursprünglich deklariert wurde. Sie können z. B. nicht

```
console.writeline("Hello World");
```

schreiben, da die `Console`-Klasse und deren `WriteLine`-Methode eben genau so deklariert wurde, wie ich diese hier beschreibe.

Anweisungsblöcke

Oft müssen mehrere Anweisungen blockweise zusammengefasst werden. Eine Funktion enthält z. B. ein Block von Anweisungen:

```
private void SayHello()
{
   string name;
   Console.Write("Ihr Name: ");
   name = Console.ReadLine();
   Console.WriteLine("Hallo " + name + ", wie gehts?");
}
```


Blöcke werden in C#, wie in C++ und in Java, mit geschweiften Klammern umschlossen. Wie in anderen Sprachen auch, kann ein Block überall dort eingesetzt werden, wo eine einzelne Anweisung erwartet wird. Die `If`-Verzweigung kann z. B. mit einzelnen Anweisungen arbeiten. Die folgende Anweisung überprüft, ob heute Sonntag ist und gibt in diesem Fall einen entsprechenden Text an der Konsole aus:

```
if (System.DateTime.Now.DayOfWeek == 0)
   Console.WriteLine("Heute ist Sonntag.");
```

Wenn für einen Fall, dass heute Sonntag ist, nun aber mehrere Anweisungen ausgeführt werden sollen, müssen Sie einen Block einsetzen:

```
if (System.DateTime.Now.DayOfWeek == 0)
{
   Console.WriteLine("Heute ist Sonntag.");
   Console.WriteLine("Heute wird nicht gearbeitet.");
   Console.WriteLine("Heute gehen wir snowboarden.");

}
```

Falls Sie die letzte Anweisung nicht verstehen, weil Sie irgendwo gelesen haben, dass ich am Niederrhein wohne: Auch am Niederrhein kann man snowboarden. In Neuss. In der Allrounder-Schneesporthalle. Sogar im Sommer ☺.

Würden Sie die Blockklammern weglassen,

```
if (System.DateTime.Now.DayOfWeek == 0)
   Console.WriteLine("Heute ist Sonntag.");
   Console.WriteLine("Heute wird nicht gearbeitet.");
   Console.WriteLine("Heute gehen wir snowboarden.");
```

würde der Compiler nur die erste Anweisung der `If`-Verzweigung zuordnen und die zweite und dritte immer ausführen, auch dann, wenn kein Sonntag ist. Ich fände das zwar recht nett (jeden Tag snowboarden ...), korrekt ist die Programmierung aber nicht.

Die Blockklammern teilen dem Compiler also mit, dass die darin enthaltenen Anweisungen zusammengehören.

Sichere und unsichere Anweisungen

Normale Anweisungen sind in C# sicher. Die CLR überprüft sichere Anweisungen daraufhin, ob diese u. U. eine Operation ausführen, die Speicherbereiche beeinflussen würde, die nicht zum aktuellen Kontext gehören. Die CLR erlaubt deswegen z. B. keine Zeiger, weil diese keine Kontrolle über den Speicherbereich ermöglichen, der über den Zeiger bearbeitet wird.

Sie können jedoch auch unsichere Anweisungsblöcke verwenden, wenn Sie diese mit dem Schlüsselwort `unsafe` kennzeichnen:

```
unsafe
{
   ...
}
```

Alternativ können Sie ganze Methoden als unsicher kennzeichnen:

```
unsafe void UnsafeDemo()
{
 ...
}
```

In einem solchen Block können Sie nun alle Anweisungen unterbringen, auch solche, die die CLR als unsicher erkennen würde. Unsichere Blöcke benötigen Sie immer dann, wenn Sie direkt auf den Speicher zugreifen wollen. Normalerweise müssen Sie dies nur tun, wenn Sie extrem schnelle Programme schreiben wollen. Mit Zeigern können Sie eben direkt mit dem Speicher arbeiten. Zeiger verursachen aber auch massive Probleme, da Sie damit versehentlich Speicherbereiche überschreiben oder auslesen können, die Sie gar nicht reserviert haben. Da dieses Thema wohl eher die C++-Programmierer interessiert, die systemnahe Programme wie z. B. Treiber entwickeln, behandle ich unsichere Programmierung in diesem Buch nicht weiter.

3.2.2 Kommentare und XML-Dokumentation

C# kennt vier Arten von Kommentaren: Einfache Kommentare, mehrzeilige einfache Kommentare, Aufgabenkommentare und Dokumentationskommentare (für eine XML-Dokumentation). Unabhängig von der Art des Kommentars kompiliert der Compiler diese nicht mit in die Assemblierung.

Kommentare im Quellcode sind in den meisten Programmen der Schlüssel zum Verständnis eines Programms. Gute Programme enthalten etwa genau so viel Kommentarzeilen wie Programmzeilen. Kommentieren Sie also immer ausführlich.

Einfache Kommentare

Einfache Kommentare enthalten lediglich einen beschreibenden Text, der das Verständnis eines Programmteils erleichtert. Einfache Kommentare können Sie an das Ende einer Zeile anfügen, indem Sie diese mit zwei Schrägstrichen einleiten:

```
i = 1; // Das ist ein einfacher Kommentar
```

Mehrzeilige einfache Kommentare werden mit /* eingeleitet und mit */ beendet:

```
Console.WriteLine("Hello User"); /* Das ist ein
   mehrzeiliger Kommentar */
```

Visual Studio beginnt die zweite Zeile eines mehrzeiligen Kommentars automatisch mit einem Stern:

```
/* Das ist ein mehrzeiliger Kommentar,
 * der mit Visual Studio
 * erzeugt wurde. */
```

Die Sterne vor den einzelnen Zeilen dienen lediglich der besseren optischen Darstellung und haben keine weitere Bedeutung.

Aufgabenkommentare

Wie ich bereits in Kapitel 2 beschrieben habe, können Sie den Text eines Kommentars automatisch in die Aufgabenliste übernehmen, wenn Sie den Kommentar mit einem Aufgabentoken (per Voreinstellung: `HACK`, `TODO`, `UNDONE`, `UnresolvedMergeConflict`) beginnen:

```
static void Main(string[] args)
{
   // TODO: Den Rest programmieren :-)
}
```

Sie sollten diese speziellen Aufgabenkommentare immer dann nutzen, wenn irgendwo im Quellcode noch irgendetwas zu tun ist. Da diese Kommentare immer automatisch in der Aufgabenliste erscheinen und Sie diese Liste auch gut filtern können (etwa nur die Kommentare anzeigen lassen), haben Sie immer eine gute Übersicht über die noch zu erledigenden Aufgaben.

Dokumentationskommentare

Ein Problem normaler Kommentare ist, dass Sie daraus keine Dokumentation erzeugen können. Dieses Problem lösen Dokumentationskommentare. Aus dem Inhalt dieser Kommentare können Sie über einen speziellen Compilerschalter bzw. über eine Visual Studio- oder SharpDevelop-Option eine XML-Datei erzeugen. Diese XML-Datei können Sie dann über die Entwicklungsumgebung oder ein externes Tool in eine professionelle HTML-Dokumentation transformieren.

Ich beschreibe hier nur die Grundlagen von Dokumentationskommentaren, weil eine komplette Beschreibung den Rahmen des Buchs sprengen würde. Auf der Website finden Sie einen Artikel, der sich mit den vordefinierten XML-Elementen und der Erzeugung einer XML- und HTML-Dokumentation mit verschiedenen Tools beschäftigt.

Dokumentationskommentare werden mit drei Schrägstrichen eingeleitet. Mit diesen Kommentaren können Sie alle Typen (Klassen, Strukturen, Aufzählungen etc.) und deren Elemente (Methoden, Eigenschaften, Ereignisse) dokumentieren. Innerhalb der Kommentare können Sie einfachen Text und XML-Elemente unterbringen. Über XML-Elemente können Sie eine eigene XML-Struktur aufbauen. Sie können eigene XML-Elemente einfügen (die Sie dann aber auch selbst in der XML-Datei auswerten müssen) oder einige der vordefinierten (die von den Tools zur Erzeugung einer HTML-Dokumentation verwendet werden). Das Element *summary* wird z. B. verwendet, um einen Typen oder ein Element zusammenfassend zu beschreiben:

```
/// <summary>
/// Der Einstiegspunkt der Anwendung
/// </summary>
[STAThread]
static void Main(string[] args)
...
```

Wenn Sie das Programm mit dem Kommandozeilencompiler kompilieren, geben Sie im Argument `/doc` an, in welche Datei die XML-Dokumentation erzeugt werden soll:

```
csc /doc:Dokumentation.xml Quellcode.cs
```

In Visual Studio geben Sie die Dokumentationsdatei in den Projekteigenschaften an. Wählen Sie dazu im Kontextmenü des Projekteintrags im Projekt-Explorer den Befehl EIGENSCHAFTEN. Stellen Sie den Dateinamen unter KONFIGURATIONSEIGENSCHAFTEN / ERSTELLEN / DOKUMENTATIONSDATEI ein. Beachten Sie, dass Sie diese Einstellung für jede Konfiguration (*Debug*, *Release*) einzeln vornehmen müssen. Wenn Sie das Projekt erstellen oder mit F5 starten, wird die Dokumentationsdatei automatisch erzeugt.

Damit Sie die HTML-Dokumentation einmal ausprobieren können, erzeugen Sie eine solche ganz einfach in Visual Studio über das Menü EXTRAS / ERSTELLEN VON KOMMENTARWEBSEITEN.

3.2.3 Der Aufruf von Methoden (Funktionen)

Methoden bzw. Funktionen enthalten, wie Sie ja bereits wahrscheinlich wissen, vorgefertigte Programme, die Sie über den Namen der Methode/Funktion aufrufen können. Da Funktionen in C# nur in Klassen organisiert werden können, werden diese auch als Methoden bezeichnet. Ich verwende also den allgemeineren Begriff.

Aufruf

Methoden werden immer mit deren Namen aufgerufen, gefolgt von Klammern, in denen Sie der Methode meist Argumente übergeben, die die Ausführung steuern. Methoden sind immer in Klassen gespeichert. Die `WriteLine`-Methode gehört z. B. zu der Klasse `Console`. Deshalb müssen Sie die Klasse immer mit angeben, wenn Sie Methoden aufrufen. Eine Ausnahme ist, wenn Sie eigene Methoden aufrufen, die in der Klasse deklariert sind, in der der Aufruf erfolgt. Wie Sie ja bereits wissen, sind Klassen in Namensräumen organisiert. Wenn Sie den Namensraum nicht über die `using`-Direktive »importiert« haben, müssen Sie dessen kompletten beim Aufruf einer Methode angeben. Die `Console`-Klasse gehört z. B. zum Namensraum `System`. Den vollständigen Aufruf der `WriteLine`-Methode zeigt der folgende Quellcode:

```
System.Console.WriteLine("Hello World");
```

Binden Sie den `System`-Namensraum über `using` ein, müssen Sie nur noch die Klasse beim Aufruf angeben:

```
Console.WriteLine("Hello World");
```

Falls Sie in der OOP schon etwas bewandert sind und sich wundern, dass z. B. die `WriteLine`*-Methode der* `Console`*-Klasse so einfach aufgerufen werden kann ohne dass Sie ein Objekt dieser Klasse erzeugen: In C# ist es, wie auch in anderen Sprachen, möglich, so genannte statische Methoden (oder auch »Klassenmethoden«) in eine Klasse zu integrieren. Solche Methoden können eben ohne Instanz der Klasse aufgerufen werden. Statische Methoden beschreibe ich noch näher in Kapitel 4.*

Argumente übergeben

Die Deklaration der Methode legt fest, wie viele Argumente welchen Typs übergeben werden müssen. In Visual Studio zeigt IntelliSense beim Schreiben einer Anweisung die Deklaration(en) der Methode und damit die zu übergebenden Argumente an (Abbildung 3.1).

```
static void Main(string[] args)
{
    Console.WriteLine(
}
```

▲1 of 19 ▼ void Console.WriteLine (**string format**, params object[] arg)
format: Format string.

Bild 3.1: Die Syntaxhilfe beim Schreiben von Anweisungen mit Methoden

Viele Methoden liegen allerdings gleich in mehreren, so genannten »überladenen« Varianten vor. Das Überladen von Methoden erläutere ich noch näher für eigene Methoden ab Seite 143, fürs Erste reicht es aus, dass Sie wissen, dass eine Methode auch in mehreren Varianten vorkommen kann. Die `WriteLine`-Methode kommt z. B. (zurzeit) in 19 Varianten vor. Die einzelnen Varianten können Sie in der Syntaxhilfe über die Cursortasten oder einen Klick auf die Pfeile in der Syntaxhilfe anzeigen lassen. Die Variante 6 erwartet z. B. nur einen einfachen String (eine Zeichenkette) als Argument (Abbildung 3.2).

```
static void Main(string[] args)
{
    Console.WriteLine(
}
```

▲6 of 19 ▼ void Console.WriteLine (**string value**)
value: The value to write.

Bild 3.2: Die Variante 6 der WriteLine-Methode

Gibt die Methode einen Wert zurück, können Sie diesen Wert in Zuweisungen, in arithmetischen oder logischen Ausdrücken oder als Argument einer anderen Methode verwenden. Stellen Sie sich einfach vor, dass der Compiler Methoden immer zuerst aufruft, bevor er weitere Teile der Anweisung auswertet und das Ergebnis der Methode dann als Wert in der Anweisung weiterverarbeitet. So sind einfache Zuweisungen möglich:

```
sinY = Math.Sin(y);
```

oder Ausdrücke:

```
result = Math.Min(x1, x2) / Math.Max(y1, y2);
```

oder geschachtelte Aufrufe:

```
result = Math.Min(y1, Math.Max(y2, y3));
```

Überall da, wo ein bestimmter Datentyp erwartet wird, können Sie neben Konstanten und Variablen immer auch eine Methode einsetzen, die diesen Datentyp zurückgibt.

3.3 Datentypen

Wie Sie ja sicherlich bereits wissen, kommen Datentypen in einem Programm sehr häufig vor. Wenn Sie einen Ausdruck schreiben, verwendet dieser einige Datentypen und besitzt im Ergebnis auch einen Datentyp. Variablen besitzen einen Datentyp, einfache Wertangaben ebenfalls, genau wie Argumente und Rückgabewerte von Methoden. Der Umgang mit Datentypen ist also sehr wichtig und wird deshalb hier grundlegend beschrieben.

C# unterstützt die üblichen Datentypen wie z. B. `int` (Integer) und `double`. Alle C#-Typen sind nur Aliasnamen für Datentypen, die in der CLR definiert sind. `int` steht z. B. für `System.Int32`. Prinzipiell können Sie immer die CLR-Typen verwenden, z. B. um eine Variable zu deklarieren:

```
System.Int32 i;
```

Verwenden Sie aber lieber die C#-Typen, damit Ihr Quellcode besser lesbar wird.