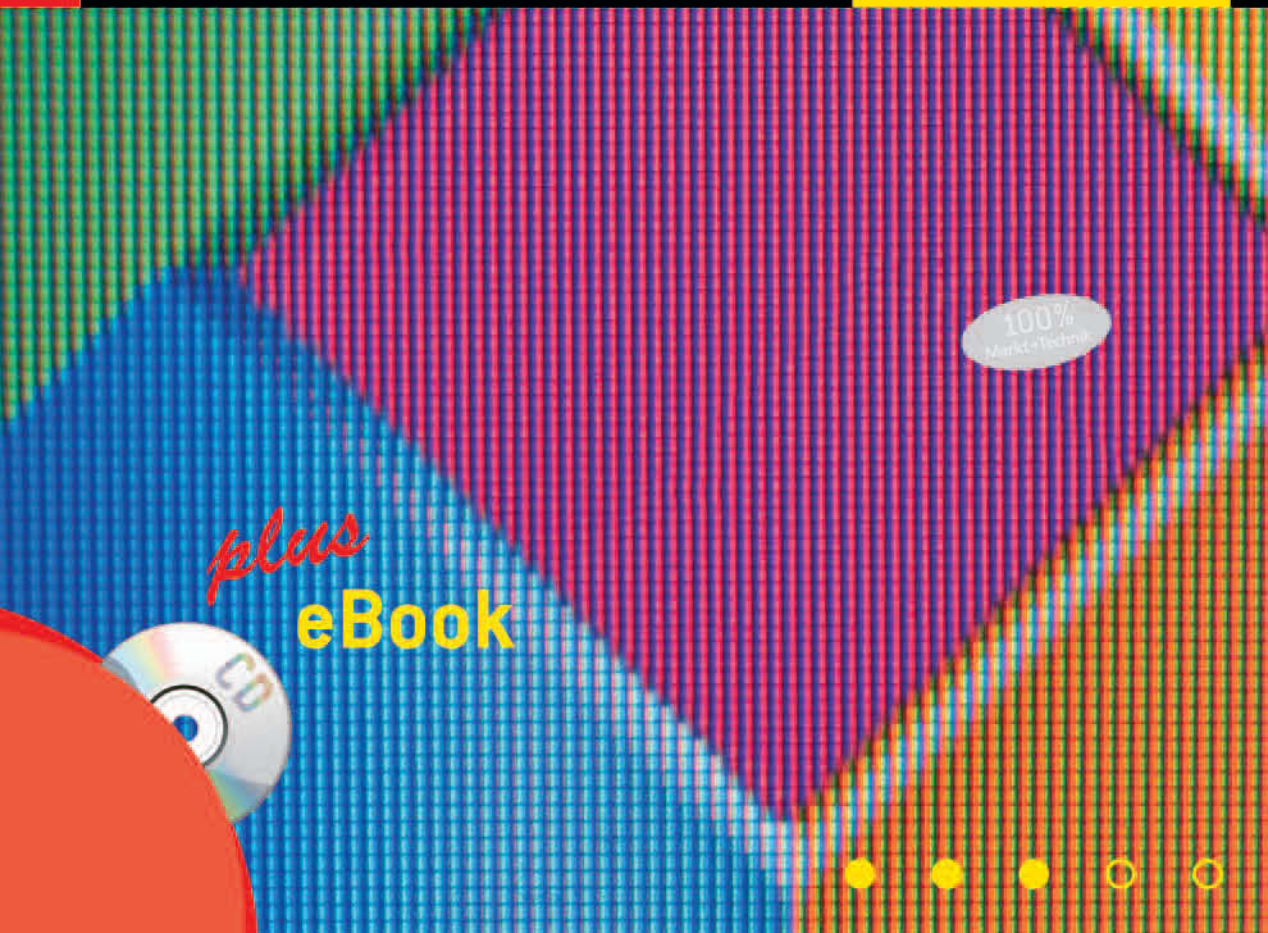
IN 14 TAGEN

Visual C# 2008

Schritt für Schritt zum Profi

DIRK LOUIS SHINJA STRASSER

Markt+Technik



Variablen und Operatoren

Alle Programme haben etwas gemeinsam – sie verwalten, speichern und verarbeiten Daten. Verwendet man diese Daten in einem Programm, müssen diese auch irgendwo gespeichert werden. Hierzu gibt es *Variablen* und *Konstanten*, mit deren Hilfe man die Daten speichern, verarbeiten und manipulieren kann. Mit Hilfe von Anweisungen und Operatoren lassen sich Programmlogiken erzeugen, die mit den gespeicherten Daten etwas berechnen oder ausführen.

Heute lernen Sie

- was eine Variable ist,
- wie man Variablen deklariert und initialisiert,
- wie und wann man die elementaren Datentypen einsetzt,
- den Unterschied zwischen impliziter und expliziter Konvertierung,
- was literale Konstanten sind und was symbolische Konstanten für einen Zweck haben,
- wofür der Aufzählungstyp gut ist,
- den Unterschied zwischen Ausdruck und Anweisung,
- welche Operatoren C# bereitstellt,
- die Einteilung der Rangfolge von Operatoren,
- wie man Quellcode am besten darstellt,
- wie die `if-else`-Anweisung arbeitet.

3.1 Was ist eine Variable?

Wie in jeder Programmiersprache dienen auch in C# Variablen zum Speichern von Informationen. Eine Variable fungiert im Speicher eines Computers als ein benannter Platzhalter bestimmter Größe, in dem Sie Informationen ablegen und wieder abrufen können.

Man unterscheidet verschiedene Variableneigenschaften, die bei der Informationsverarbeitung in einem Programm relevant sind. Dies betrifft grundsätzlich drei Aspekte:

- Typ der Variablen
- gespeicherte Daten
- Name der Variablen

Der Typ

Der Typ einer Variablen legt die Größe fest, die für die Daten im Speicher reserviert wird. Zusätzlich bestimmt der Typ die Art und Weise der Daten, die in dieser Variablen abgelegt werden dürfen und können.

Das bedeutet, dass die Informationen, die in der Variablen abgelegt werden sollen, durch die Typdefinition validiert (evtl. auch konvertiert und verfälscht) werden. Wenn Sie zum Beispiel eine Variable vom Typ `int` (Ganzzahl, engl. *integer*) verwenden, können Sie keine Daten im Format `3.14` in dieser Variablen speichern. Der Typ `int` repräsentiert in C# die Ganzzahlen des Zahlensystems und die Zahl `3.14` stellt eben eine Gleitkommazahl dar.

```
int iGanzzahl;

iGanzzahl = 3.14;
```

Der Compiler meldet hier einen Fehler, der folgendermaßen lautet:

```
Implizite Konvertierung des Typs 'double' zu 'int' nicht möglich.
```

Eine Möglichkeit, Daten verschiedener Typen miteinander zu kombinieren, besteht darin, diese durch eine Konvertierungsanweisung umzuwandeln. Hierbei muss man aber mit einer Verfälschung bis hin zu einem teilweisen Datenverlust der bestehenden Information rechnen. Wie man einen Typ einer Variablen umwandelt, wird später noch genauer beschrieben (siehe Abschnitt 3.5).

Die Daten

Die Daten stellen den Inhalt einer Variablen dar. Daten besitzen immer ein bestimmtes Format, das natürlich mit dem Typ einer Variablen übereinstimmen muss, damit Daten in der Variable gespeichert werden können. Die Daten, die Sie in einer Variablen ablegen, stehen in engem Zusammenhang mit dem Typ einer Variablen. Das nächste Beispiel zeigt ein typisches Formatierungsproblem.

```
int iGanzzahl;

iGanzzahl = 314.0;
```

Die Zahl `314.0` stellt an sich eine Ganzzahl dar und jeder Taschenrechner würde die Nachkommazahl `.0` ignorieren. Die Programmiersprache C# nimmt jedoch die Typisierung der Daten sehr streng. Für den C#-Compiler bedeutet das, dass er die Zahl `314.0` als Gleitkommazahl interpretiert und auch hier die gleiche Fehlermeldung ausgibt:

```
Implizite Konvertierung des Typs 'double' zu 'int' nicht möglich.
```

Der Name

Der Name einer Variablen identifiziert eindeutig den reservierten Speicher, um auf die abgelegten Daten zugreifen zu können. Er dient sozusagen als Synonym eines Adressbereichs im Speicher. Wie man die Namensgebung sinnvoll gestaltet, wird anschließend noch genauer besprochen. Der Name kann aus Buchstaben, Zahlen und dem Zeichen »_« (Unterstrich) zusammengesetzt werden. Das erste Zeichen des Namens muss ein Buchstabe oder ein Unterstrich sein.

3.1.1 Der Variablenname – die Qual der Wahl

Die richtige Wahl des Namens einer Variablen ist nicht immer so einfach, wie es auf den ersten Blick erscheint. Variablennamen sollten nach Möglichkeit so kurz wie möglich gehalten werden, aber das Maximum an Aussagekraft haben. Häufig werden in Softwarefirmen eigene Konventionen verwendet, um Variablen zu benennen. Aber auch ohne Konventionen sollten Sie sich Gedanken hinsichtlich der Benennung von Variablen machen. Dieser Abschnitt soll Ihnen einen kleinen Leitfaden zur Verfügung stellen, um Variablen sinnvoll zu benennen.

In der Praxis üblich sind Namensschemata, bei denen die Variablennamen mit einer Typinformation beginnen. So könnte einer Variablen vom Typ `string` der Buchstabe `s` oder die Zeichenfolge `str` vorangestellt werden.

Diese Typinformationen erleichtern dem Programmierer die Zuordnung der Datentypen zu den einzelnen Variablen. Wird diese Namensgebung konsequent durchgeführt, kann der Programmierer den Quellcode besser und schneller verstehen, da nicht ständig nach der Variablendeklaration gesucht werden muss. Die Tabelle 3.1 zeigt Ihnen einige Beispiele, wie Sie Typinformationen in eine Variable einbringen können.

Datentyp	Präfix
`boolean`	`bln`, `b` (`bAnrede`)
`byte`	`byt`, `bt` (`btMonat`)
`sbyte`	`sbyt`, `sbt` (`sbtWoche`)
`double`	`dbl`, `d` (`dWaehrung`)
`int`	`int`, `i` (`iAlter`)
`long`	`lng`, `l` (`lSekunden`)
`object`	`obj`, `o` (`oMyObject`)

Tabelle 3.1: Präfixe für die wichtigsten Datentypen

Vermeiden Sie unbedingt Namen, die die gleiche Bedeutung haben, aber verschieden geschrieben werden. Solche Beispiele wären:

`iMwSt`, `iMWST`, `iMwST`, ...

Stilkonventionen

Es gibt verschiedene Konventionen für die Schreibweise von Variablen. Je nach Gewohnheit findet man folgende Schreibweisen in einem Programm

```
int iNachSteuer;
int inachsteuer;
int inach_steuer;
```

Konstanten (Variablen, deren Werte nach der Zuweisung nicht mehr veränderbar sind[1]) werden in Großbuchstaben geschrieben.

```
const int MWST = 16;
```

Wenn man ein fremdes Programm weiterentwickelt, empfiehlt es sich die Schreibweise des Vorgängers beizubehalten[2].

1 Der Begriff *Konstante* sagt vom Wortlaut eigentlich, dass es sich um das Gegenteil einer Variablen handelt, eben um eine nicht variable Größe. Sie sollten sich nicht daran stören, es hat sich eingebürgert, eine Konstante als eine besondere Form einer Variablen zu betrachten.

Die allgemeine Konvention für die Benennung aller Elemente (Variablen, Klassen, ...) einer Programmiersprache werden meist durch die folgenden Schreibweisen festgelegt.

Pascal Case: Diese Schreibweise verwendet man für fast alle Elemente der Programmiersprache C#. Jedes Wort fängt im *Pascal Case* mit einem Großbuchstaben an. `ToString` (z. B. für eine Methode), `iBackColor` (für Variablen [mit Typpräfix, daher erster Buchstabe klein]) usw.

Camel Case: Diese Schreibweise verwendet man für Parameter und private Klassenelemente. Das erste Wort beginnt mit einem Kleinbuchstaben, die folgenden Wörter mit einem Großbuchstaben: `typeName`, `refValue` usw.

Upper Case: Diese Schreibweise verwendet man ausschließlich für Konstanten. Alle Buchstaben werden hier groß geschrieben: `MWST`, `PI` usw.

3.1.2 Variablendeklaration

Wir wissen nun, dass eine Variable grundsätzlich drei miteinander eng verbundene Eigenschaften besitzt – Typ, Daten und Name.

Bevor man mit einer Variablen in einem Programm arbeiten kann, muss diese deklariert werden. Die Anweisung einer Deklaration kann folgendermaßen aussehen.

```
int iZahl;
```

Diese Anweisung deklariert eine Variable mit dem Namen `iZahl`, in der Daten vom Typ `int` (Ganzzahl) abgelegt werden können.

In der Literatur kann man sowohl von Variablendeklaration als auch Variablendefinition lesen. Lassen Sie sich dadurch nicht verwirren. Die Aufgabe der *Variablendeklaration* ist es, die Variable über ihren Namen dem Compiler bekannt zu machen, während im Zuge der *Variablendefinition* der Speicher für die Variable reserviert wird. In C# gehen beide Vorgänge Hand in Hand, sodass die Begriffe durchaus synonym verwendet werden können.

2 Welche Schreibweise Sie wählen hängt prinzipiell von den firmeninternen oder eigenen Konventionen ab. Mit der Zeit werden Sie Ihren eigenen Stil entwickeln, der für Sie oder Ihre Firma die effektivste Codedarstellung verkörpert. Sollten Sie einmal in die Lage kommen, Drittkomponenten zu entwickeln, sollten Sie sich an den empfohlenen Standard Pascal, Camel und/oder Upper Case halten.

Der dritte Aspekt betrifft die Eigenschaft der Daten. Wird einer Variablen ein Wert zugewiesen, spricht man von *Variableninitialisierung*.

Dass es solche Unterschiede in der Terminologie gibt, liegt daran, dass man Programme in der Regel zeilenweise liest und – wie Sie gleich sehen werden – die Variablendeklaration und die Variablendefinition (evtl. auch die Variableninitialisierung) in einer Anweisung vorgenommen werden.

Eine Variablendeklaration hat folgende Syntax

```
Typbezeichner Variablenname;
```

also z. B.:

```
int iAlter;
```

Der Typbezeichner spezifiziert den Typ der Variablen. In den folgenden Abschnitten werden Sie mehr über die Typen von C# erfahren.

Mit einem Initialisierungswert sieht die Syntax folgendermaßen aus:

```
Typbezeichner Variablenname = Initialisierungswert;
```

also z. B.:

```
int iAlter = 0;
```

Beachten Sie, dass eine Variablendeklaration immer mit einem Semikolon endet, wie im Allgemeinen alle Anweisungen in C#.

Neben Deklarationen einzelner Variablen können auch mehrere Variablen innerhalb einer Anweisung deklariert werden, man spricht hier von Mehrfachdeklaration. Hierbei schreibt man nach dem Typbezeichner mehrere Variablennamen und trennt diese mit einem Komma, wie folgendes Beispiel zeigt:

```
int iZahl1, iZahl2, iErgebnis;
```

oder auch

```
int iZahl1 = 0, iZahl2 = 0, iErgebnis = 0;
```

Hier werden drei Variablen `iZahl1`, `iZahl2` und `iErgebnis` vom Typ `int` deklariert bzw. mit dem Ganzzahlwert `0` initialisiert.

Sehen wir uns jetzt einmal anhand eines Beispiels an, wie die Variablendeklaration verwendet wird, und diskutieren die Eigenschaften und Eigenheiten von deklarierten Variablen. Das Programm gibt die deklarierten Variablen in der Konsole aus. Geben Sie folgendes Listing in Ihren Editor ein, kompilieren Sie es und führen es aus.

Listing 3.1: Variablendeklaration.cs

```
01: using System;
02: namespace Kap3
03: {
04:    public class CVar
05:    {
06:       static void Main(string[] args)
07:       {
08:         int iZahl1 = 0, iZahl2;
09:         int iErgebnis = 7;
10:
11:         iZahl2 = 0;
12:         Console.WriteLine("1.Zahl: {0}", iZahl1);
13:         Console.WriteLine("2.Zahl: {0}", iZahl2);
14:         Console.WriteLine("Ergebnis: {0}", iErgebnis);
15:
16:         iZahl1 = 2;
17:         iZahl2 = 8;
18:
19:         Console.WriteLine("1.Zahl: {0}", iZahl1);
20:         Console.WriteLine("2.Zahl: {0}", iZahl2);
21:         Console.WriteLine("Ergebnis: {0}", iZahl1 + iZahl2);
22:       }
23:    }
24: }
```

```
1.Zahl: 0
2.Zahl: 0
Ergebnis: 7
1.Zahl: 2
2.Zahl: 8
Ergebnis: 10
```

Die Zeilen 1 bis 7 kennen wir ja schon aus dem ersten Kapitel. Beginnen wir mit Zeile 8. Hier werden die beiden Variablen `iZahl1` und `iZahl2` deklariert und die Variable `iZahl1` gleichzeitig mit dem Ganzzahlwert `0` initialisiert. Es handelt sich hier um eine Mehrfachdeklaration, da die beiden Variablen in einer Zeile mit dem Typ `int` definiert werden. In der Zeile 9 wird die Variable `iErgebnis` deklariert und mit dem Wert `7` initialisiert.

Die Zeile 11 zeigt einen anderen Weg einer Variableninitialisierung, nämlich eine Initialisierung außerhalb der Deklarationsanweisung. Bevor man eine Variable in einem Programm verwenden kann, muss man diese initialisieren, da sonst der Compiler eine Fehlermeldung ausgibt.

In den Zeilen 12 bis 14 werden die Variablen ausgelesen und in der Konsole ausgegeben. Bis jetzt sind die Inhalte der Variablen klar. `iZahl1` und `iZahl2` enthalten den Wert `0` und die Variable `iErgebnis` den Wert `7`. In den Zeilen 16 und 17 weisen wir den Variablen `iZahl1` und `iZahl2` einen neuen Wert zu. In der Variablen `iZahl1` wird der Wert `2` abgelegt und in der Variablen `iZahl2` der Wert `8`. Bei diesem Vorgang werden die ursprünglichen Werte für die beiden Variablen überschrieben. Grundsätzlich besteht zwischen der Zeile 11 und den Zeilen 16, 17 kein Unterschied. In beiden Fällen werden den Variablen Werte zugewiesen. Wird aber einer Variablen zum ersten Mal ein Wert zugewiesen, spricht man von Initialisieren einer Variablen.

In den Zeilen 19 und 20 werden die Werte von `iZahl1` und `iZahl2` ausgegeben und in Zeile 21 die Summe der beiden Inhalte der Variablen. Die Variable `iZahl1` enthält den Wert `2`, die Variable `iZahl2` den Wert `8` und die Variable `iErgebnis` den unveränderten Wert `7`.

Versuchen Sie einmal, das folgende Programm auszuführen:

Listing 3.2: Variablen_nichtinitialisiert.cs – Fehlen der Initialisierung

```
01: using System;
02: namespace Kap3
03: {
04:    public class CVar
05:    {
06:       static void Main(string[] args)
```

```
07:       {
08:          int iZahl1 = 0, iZahl2;
09:          int iErgebnis = 7;
10:
11:         Console.WriteLine("1.Zahl: {0}", iZahl1);
12:         Console.WriteLine("2.Zahl: {0}", iZahl2);
13:         Console.WriteLine("Ergebnis: {0}", iErgebnis);
14:
15:         iZahl1 = 2;
16:         iZahl2 = 8;
17:
18:         Console.WriteLine("1.Zahl: {0}", iZahl1);
19:         Console.WriteLine("2.Zahl: {0}", iZahl2);
20:         Console.WriteLine("Ergebnis: {0}", iZahl1 + iZahl2);
21:       }
22:    }
23: }
```

Sie sehen, dass die Variable `iZahl2` nicht initialisiert wird und der Compiler die folgende Fehlermeldung ausgibt:

```
error CS0165: Verwendung von möglicherweise nicht zugewiesenen lokalen
Variablen 'iZahl2'
```

Der C#-Compiler lässt es nicht zu, Variablen zu verwenden, die nicht initialisiert wurden.

3.2 Schlüsselwörter

Schlüsselwörter sind vordefinierte Wörter, mit denen bestimmte Aktionen verbunden sind. Diese Wörter sind also innerhalb des Sprachumfangs von C# reserviert. Die Tabelle unten zeigt eine Auflistung der Schlüsselwörter in C#.

abstract	enum	long	stackalloc
as	event	namespace	static
base	explicit	new	string
bool	extern	null	struct

Tabelle 3.2: Schlüsselwörter von C#

break	false	object	switch
byte	finally	operator	this
case	fixed	out	throw
catch	float	override	true
char	for	params	try
checked	foreach	private	typeof
class	goto	protected	uint
const	if	public	ulong
continue	implicit	readonly	unchecked
decimal	in	ref	unsafe
default	int	return	ushort
delegate	interface	sbyte	using
do	internal	sealed	virtual
double	is	short	void
else	lock	sizeof	while

Tabelle 3.2: Schlüsselwörter von C# (Forts.)

In C# gibt es auch eine Möglichkeit, reservierte Wörter als Variablennamen zu verwenden. In der Regel sollten Sie aber darauf verzichten, da dies nicht besonders elegant ist. Wenn es doch einmal gewünscht ist, müssen Sie dem Variablennamen ein @-Zeichen voranstellen.

```
01: using System;
02:
03:  namespace Demo
04:  {
05:
06:    class CDemo
07:    {
08:       static void Main(string[] args)
09:       {
10:         decimal @decimal = 10.0m;
11:
```

```
12:          Console.WriteLine("{0}",@decimal);
13:
14:          Console.ReadLine();
15:        }
16:    }
17: }
```

In Zeile 10 wird eine Variable `@decimal` deklariert. Der Name `decimal` alleine führt zu einer Fehlermeldung des Compilers.

```
10:          decimal decimal = 10.0m;
```

```
Identifizierer erwartet, 'decimal' ist ein Schlüsselwort
```

`Console.ReadLine()` liest eine Zeile aus der Konsole ein, Sie werden also bei der Programmausführung aufgefordert, einen Text einzugeben. Hier wird die Methode `ReadLine()` verwendet, um das Ergebnis in der Konsole sehen zu können. Ohne Verwendung der Methode `ReadLine()` würde sich die Konsole sofort nach der Ausführung des Programms schließen. (Diesen Trick werden Sie noch häufiger in diesem Buch finden; er bewirkt, dass die Konsole sichtbar bleibt, wenn Sie mit Visual C# arbeiten und dort `Debuggen/Starten` wählen. Bei Einsatz des Kommandozeilen-Compilers *csc* ist der Trick nicht notwendig – Sie können also `ReadLine()` weglassen. Auch bei Visual C# können Sie auf `ReadLine()` verzichten, wenn Sie das Programm mit DEBUGGEN/STARTEN OHNE DEBUGGING ausführen).

3.3 Elementare Datentypen

Bevor wir uns mit den elementaren Datentypen der Programmiersprache C# beschäftigen, machen wir einen kleinen Ausflug in die Welt der Bits und Bytes. Wenn Sie ein Programm schreiben und es ausführen, passiert dies im Arbeitsspeicher des Computers. Zu Beginn des Buches, wo wir über den Garbage Collector sprachen, behauptete ich, dass wir uns über die Organisation und die Verwaltung des Speichers keine Gedanken machen müssen. Um aber als Programmierer entscheiden zu können, welchen Datentyp man wählen soll, ist ein Verständnis der inneren Struktur einer Variablen von großer Bedeutung.

Wenn Sie in C# Informationen speichern wollen, so machen Sie das, indem Sie eine Variable deklarieren. Diese besitzt nach der Deklaration einen Namen und einen bestimmten Typ. Die Variable beansprucht Platz im Arbeitsspeicher des Computers, wo das Programm ausgeführt wird. Je nach Typ ist der beanspruchte Platz größer oder kleiner. Was aber genau ist dieser Platz im Arbeitsspeicher eines Computers?

Beginnen wir mit der Einheit des Speichers eines Computers. Jeder Computer hat einen Arbeitsspeicher, den man sich als eine sequentielle Anordnung von Quadern vorstellen kann. Einen solchen Quader in einem Arbeitsspeicher bezeichnet man als Byte. Um aber zu wissen, welches Byte man ansprechen soll, müssen diese Bytes individuell gekennzeichnet werden. Diese Kennzeichnung eines Bytes nennt man Adresse. Jedes Byte in einem Arbeitsspeicher besitzt seine eigene Adresse. Wenn Sie in C# mit verwaltetem Code (*managed code*) arbeiten, brauchen Sie sich über die Adressen im Arbeitsspeicher keine großen Gedanken machen. Die Operationen, die mit diesen Adressen vollzogen werden, übernimmt im verwalteten Code der Garbage Colletor.

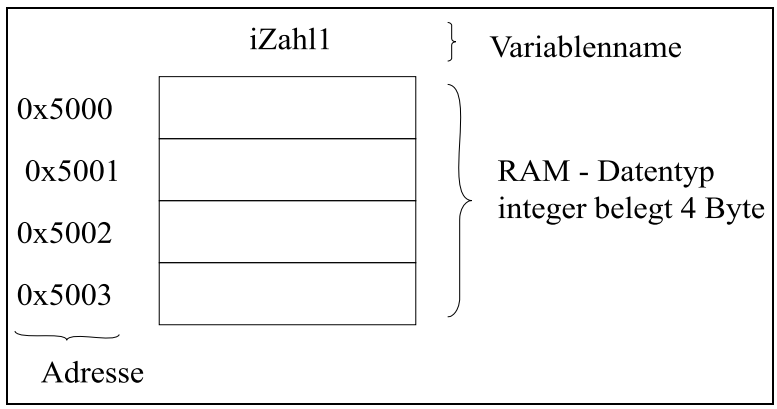


Abbildung 3.1: Aufbau des RAM

Wenn wir uns ein Byte aus dem Arbeitsspeicher genauer anschauen, müssen wir noch eine weitere Einteilung vornehmen. Ein Byte besteht aus 8 Bit. Ein Bit ist die kleinste Einheit des Arbeitsspeichers und kann genau zwei Werte annehmen, entweder eine logische 0 oder eine logische 1.

Wie erkennt man dann aber eine Ganzzahl oder eine Zeichenkette, wenn im Speicher nur Einsen und Nullen existieren? Das möchte ich Ihnen im nächsten Abschnitt anhand einer Integervariablen zeigen.

3.3.1 Der integer-Typ

Jedes der 8 Bit, die zusammen ein Byte ergeben, hat eine Wertigkeit basierend auf dem Positionssystem mit der Basis 2. Das Bit ganz rechts steht für 2^0 (dezimal 1), das nächste für 2^1 (dezimal 2), das dritte von rechts für 2^2 (dezimal 4) usw. Um den dezimalen Wert eines Bytes zu ermitteln, addiert man die einzelnen Zweierpotenzen, wobei nur gesetzte Bits (1) addiert werden, ungesetzte Bits (0) werden ignoriert.

Hier ein kleines Beispiel mit dem Dezimalwert 7, dieser wird binär folgendermaßen dargestellt:

```
00000111
```

Dies lässt sich leicht nachweisen, indem man die Zweierpotenzen aufaddiert:

$$X = 0*2^7+0*2^6+0*2^5+0*2^4+0*2^3+1*2^2+1*2^1+1*2^0$$

Daraus ergibt sich 0+0+0+0+0+4+2+1, also 7.

Sie sehen, dass dies recht trivial ist und dass man mit dieser Systematik relativ einfach jede Dezimalzahl in das Binärformat und jeden Binärwert in eine dezimale Zahl umwandeln kann. Hier einige weitere Beispiele:

```
128,64,32,16,8,4,2,1
  0  0  0  0 0 0 0 0    =  0
  0  1  0  0 0 0 0 1    =  65
  0  0  1  0 1 1 0 1    =  45
  1  1  1  1 1 1 1 1    =  255
```

Da ein Byte aus 8 Bit besteht, ergeben sich 256 (2^8) verschiedene Werte, also von dezimal 0 bis 255.

Jetzt wissen wir, wie wir die Bits und Bytes interpretieren müssen. Im Grunde steht uns nichts mehr im Weg, den Datentyp `int` zu analysieren. Um die Größe der Datentypen feststellen zu können, wechseln wir in den `unsafe`-Mode. Das bedeutet, dass dieser Code von der CLR nicht nachprüfbar ist und auch die automatische Speicherverwaltung durch den Garbage Collector keinen Einfluss auf das Programm hat.

Folgendes Beispiel gibt die Anzahl der Bytes zurück, die der Datentyp Integer im Arbeitsspeicher reserviert. Geben Sie das Beispiel ein und kompilieren es danach folgendermaßen:

```
csc intType.cs /unsave
using System;
```

```
namespace Kap3
{
   public class CTyp
   {
      static unsafe void Main()
      {
         Console.WriteLine("Integer: {0} Bytes", sizeof(int));
      }
   }
}
```

```
Integer: 4 Bytes
```

Jetzt wissen wir, wie viele Bytes für den Typ `int` im Arbeitsspeicher reserviert werden. Vier Bytes entsprechen 4 * 8 = 32 Bit.

Jetzt stellt sich die Frage nach dem Wertebereich des Datentyps `int`. 32 Bit ergibt dezimal 2^{32} = 4.294.967.296, also etwas über 4 Milliarden; trotzdem können Sie keine Werte über 2.147.483.647 – also etwa 2 Milliarden – speichern, da der Wertebereich bei diesem Datentyp in einen jeweils gleich großen positiven und negativen Wertebereich aufgeteilt ist. Konkret bedeutet dies, dass Sie in Verbindung mit dem `int`-Datentyp Dezimalwerte von -2.147.483.648 bis +2.147.483.647 speichern können.

In C# können Sie auch noch weitere Integerdatentypen verwenden, je nach Anwendung und Anforderung. Konkret bietet die Familie der Integerdatentypen noch folgende Typen:

- `short` - System.Int16 (2^{16} Bit)
- `long` - System.Int64 (2^{64} Bit)

Wie wir bereits wissen, wird jede Programmiersprache in den IL-Code kompiliert. Der Datentyp `short` (in C#) entspricht somit dem Datentyp `System.Int16` (in der Common Language Runtime). Einfach ausgedrückt: Der C#-Datentyp `short` entspricht im IL-Code dem CLR-Datentyp `System.Int16`. Das gleiche gilt für den Datentyp long (entspricht dem CLR-Datentyp `System.Int32`).

In manchen Anwendungsfällen benötigen Sie nur den positiven Bereich des Datentyps `int`. Hier gibt es die Möglichkeit, den negativen Bereich des `int`-Datentyps auszuschließen und stattdessen den Bereich der positiven Zahlen auszuweiten. Die Deklaration einer solchen Variablen sieht folgendermaßen aus:

```
uint uiZahl = 0;
```

Das `u` vor dem `int` teilt dem Compiler mit, dass es sich hier um einen *unsigned integer*, also einen vorzeichenlosen Integerwert handelt. Der Wertebereich liegt zwischen 0 und 4.294.967.295.

3.3.2 Der short-Typ

Im vorherigen Abschnitt erwähnte ich, dass die Familie der Integer-Datentypen nicht nur aus dem 32Bit-Integer-Datentyp besteht, sondern dass noch zwei weitere Integer-Datentypen existieren.

Auch wenn im Allgemeinen ausreichend Arbeitsspeicher zur Verfügung steht und es scheinbar keine große Rolle spielt, ob ein paar Bytes eingespart werden, sollte man sich als Programmierer dennoch Gedanken machen, welche Datentypen wirklich benötigt werden. Läuft Ihr Programm z. B. auf einem Server, auf dem noch eine Vielzahl von anderen Programmen betrieben werden, wirkt sich eine Speicherverschwendung irgendwann negativ auf die Performance aus.

Der Datentyp `short` reserviert einen Bereich von 2 Bytes im Speicher des Computers, das entspricht 16 Bit (also dezimal 2^{16} = 65.536 verschiedene Werte). Für manche Variablen genügt ein `short`-Integer. Eine Verwendung des `int`-Datentyps ist mehr oder weniger eine Verschwendung von Ressourcen, wenn der Wertebereich eines `short`-Datentyps ausreicht. Auch hier gibt es mit `ushort` eine vorzeichenlose Variante.

Der Wertebereich des `short`-Datentyps erstreckt sich von –32.768 bis 32.767, der eines `ushort` von 0 bis 65.535.

3.3.3 Der long-Typ

Sollten Sie einmal in die Lage kommen, dass ein `int`-Datentyp Ihren benötigten Wertebereich nicht mehr abdeckt, so gibt es einen weiteren Datentyp – `long`. Dieser Datentyp beansprucht 8 Byte des Arbeitsspeichers, das entspricht 64 Bit. Damit können Sie 2^{64} verschiedene Werte darstellen, das entspricht etwa 18 Trillionen. Auch hier gibt es wieder den vorzeichenlosen Datentyp: `ulong`. Der Wertebereich von einem `long`-Datentyp reicht von –9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807, der eines `ulong` von 0 bis 18.446.744.073.709.551.615.

3.3.4 Der byte-Typ

In speziellen Fällen, in denen Sie einen sehr kleinen Wertebereich benötigen und möglicherweise der Datentyp `short` immer noch zu groß ist, können Sie auf den Datentyp `byte` ausweichen, der 2^8=256 Werte (0-255) darstellen kann. Möchten Sie auch noch auf den negativen Bereich zugreifen, können Sie eine Variable mit `sbyte` deklarieren. Der Datentyp `sbyte` besitzt einen Wertebereich von –128 bis 127. Die Verwendung dieses Datentyps soll das folgende Listing verdeutlichen.

```
using System;

namespace Kap3
{
   public class CByte
   {
      static void Main()
      {
         byte a = 255;
         sbyte b = -128;

         Console.WriteLine("{0}",a);
         Console.WriteLine("{0}",b);
         Console.ReadLine();
      }
   }
}
```

3.3.5 Der character-Typ

In einer Variablen, die mit dem Datentyp `char` deklariert wurde, können Sie ein einzelnes Zeichen (engl. *character*) ablegen. Ein solches Zeichen kann sowohl ein Buchstabe wie a, b, c usw. sein als auch ein Fragezeichen (?) oder das @-Zeichen. Der Datentyp `char` reserviert 2 Byte im Arbeitsspeicher – das entspricht 16 Bit.

Im Grunde genommen wird ein Zeichen als eine Abfolge von 0 und 1 gespeichert – wie zum Beispiel auch beim `short`-Datentyp. Genauso wird es hier auch gemacht. Der Computer an sich kennt keine Zeichen, sondern nur Zahlen. Jedes einzelne Zeichen wird durch eine individuelle Zahl bzw. Nummer repräsentiert.

Für die Verwaltung dieser Zeichen als Zahlen wurde ein Standard entwickelt. Diesen Standard bezeichnet man als Unicode. Über eine Unicode-Tabelle kann man die Zeichen und die zugehörigen Zahlen ermitteln. Wie aber erkennt der Compiler, ob es sich hier um eine Zahl oder um ein Zeichen handelt? Dies geschieht natürlich durch den Typ der Variablen, in der ein Zeichen abgelegt wird. Die Deklaration eines `char`-Datentyps sieht folgendermaßen aus:

```
char cLetter = 'A';
```

Das Zeichen `A` wird durch die Zahl 65 im Unicode repräsentiert. Der Typ der Variablen `cLetter` ist ein `char` und somit keine Zahl.

Da man nicht immer alle Unicode-Zeichen im Kopf haben kann, hilft Ihnen das folgende Listing nach der Suche der Zeichen bzw. der repräsentativen Zahl.

Listing 3.3: Variablen_Unicode.cs

```
01: using System;
02:
03: namespace Kap3
04: {
05:    public class CUnicode
06:    {
07:       static void Main()
08:       {
09:         int iChar = 64;
10:         char cChar = 'X';
11:
12:         Console.WriteLine("{0}({1})",(char)iChar,iChar);
13:         Console.WriteLine("{0}({1})",cChar,(int)cChar);
14:
15:         Console.ReadLine();
16:       }
17:    }
18: }
```

```
@(64)
X(88)
```

In Zeile 9 wird eine Variable `iChar` vom Typ `int` deklariert und mit dem Wert `64` initialisiert. Der Unicode `64` repräsentiert das `@`-Zeichen. In der Zeile 10 wird eine Variable vom Typ `char` deklariert und mit dem

Großbuchstaben `X` initialisiert. Um herauszufinden, welche Zahl welches Zeichen repräsentiert und umgekehrt, können Sie die Anweisungen in den Zeilen 12 und 13 verwenden und entsprechend ändern. In der Zeile 12 wird der Inhalt der Variablen `iChar` mit dem Wert `64` in das zugehörige Zeichen umgewandelt und in der Zeile 13 wird das Zeichen in die zugehörige Zahl umgewandelt. (Die vorangestellten speziellen Cast-Operatoren `(char)` und `(int)` in den Zeilen 12 und 13 nehmen die eigentliche Konvertierung vor; mehr über derartige Datentypkonvertierungen erfahren Sie in Abschnitt 3.5.)

3.3.6 Der float-Typ

Bislang haben wir uns ausschließlich mit Ganzzahlen beschäftigt. In der Praxis werden jedoch auch gebrochene Werte benötigt. Hierfür stellt C# mehrere Datentypen zur Verfügung. Einer davon ist `float`.

Der Datentyp `float` reserviert 4 Byte (32 Bit) im Arbeitsspeicher. Man könnte erwarten, dass man 2^{32}, also rund 4 Milliarden verschiedene Werte darstellen kann, was betreffend auf die Verwaltung im Arbeitsspeicher auch zutrifft. In der Praxis sieht es jedoch etwas anders aus: Die interne Speicherung ist recht komplex, so wird eigentlich eine Ganzzahl gespeichert (die so genannte Mantisse), obwohl gebrochene Werte dargestellt werden. Die Position des Kommas innerhalb dieser Ganzzahl (daher auch der Begriff *float*, da das Komma sozusagen fließt, gleitet) wird durch eine weitere Ganzzahl repräsentiert, den so genannten *Exponenten*.

An dieser Stelle soll nicht weiter darauf eingegangen werden, es sollte genügen zu wissen, dass `float` einen dezimalen Wertebereich von $1.5*10^{-45}$ bis $3.4*10^{38}$ darstellen kann. Das bedeutet, es können extrem kleine Werte dargestellt werden (positive und negative) als auch extrem große (wiederum positive und negative) . Der negative Exponent gibt an, dass das Dezimalkomma entsprechend nach links wandert, so entspricht $1.5*10^{-45}$ der sehr kleinen Zahl von

+/-0.00000000000000000000000000000000000000000000015

Analog dazu gibt ein positiver Exponent an, dass das Kommazeichen nach rechts wandert, so entspricht $3.4*10^{38}$ der gewaltigen Zahl von

+/-340 000 000 000 000 000 000 000 000 000 000 000 000

Bei Variablen vom Typ `float` müssen Sie unbedingt darauf achten, dass Sie das Spezifizierungs-Suffix `f` oder `F` verwenden (bei einem Spezifizierer spielt die Groß- und Kleinschreibung generell keine Rolle, Sie können also wahlweise einen Klein- oder Großbuchstaben schreiben). Die Zahl 5.5 bezeichnet man als echtes Literal[3], und damit wird in C# standardmäßig die literale Konstante 5.5 als Datentyp `double` deklariert (der Typ `double` wird im nächsten Abschnitt beschrieben).

```
float fPreis = 5.5;
```

Hier erhalten Sie vom Compiler folgende Fehlermeldung

```
Literale des Typs "Double" können nicht implizit zum Typ 'float'
konvertiert werden. Verwenden Sie ein 'F'-Suffix, um eine Literale mit
diesem Typ zu erstellen.
```

Um diesen Fehler zu vermeiden, müssen Sie Variablen vom Typ `float` mit den `f`-Suffix deklarieren, also das `f` hinter den Wert schreiben:

```
float fPreis = 5.5f;
```

Der sehr große Wertebereich des `float`-Datentyps sollte nicht darüber hinwegtäuschen, dass die Genauigkeit lediglich 7 Stellen beträgt, was mit der speziellen internen Verwaltung zu tun hat. Das bedeutet, wenn Sie den Wert

```
5.1111111111111111111111111111111111111111111111
```

speichern und wieder ausgeben, wird nur

```
5,111111
```

angezeigt und bei einer Speicherung von

```
5555.1111111111111111111111111111111111111111111111
```

wird

```
5555,111
```

ausgegeben, also jeweils 7 Dezimalstellen. In beiden Fällen hat der Wert an Genauigkeit verloren. Wertebereich und Genauigkeit sind damit etwas grundsätzlich Verschiedenes. Falls die Genauigkeit von 7 Stellen nicht ausreicht, müssen Sie auf den `double`-Typ ausweichen.

3 Ein echtes Literal (auch als natürliches Literal oder *Literalkonstante* bezeichnet) ist ein direkt in den Quellcode geschriebener Wert, z. B. 5.5 oder "a" im Gegensatz zu einem Wert, der durch einen Namen einer Variablen oder Konstanten repräsentiert wird.

3.3.7 Der double-Typ

Der Datentyp `double` belegt 8 Byte im Arbeitsspeicher. Somit werden für eine Gleitkommazahl 8*8=64 Bit reserviert. Die interne Verwaltung ist ähnlich wie beim `float`-Typ, jedoch ergeben sich durch die Tatsache, dass doppelt so viel Speicher verwendet wird, ein erheblich größerer Wertebereich und eine deutlich höhere Genauigkeit.

Der Wertebereich erstreckt sich von $+/-5*10^{-324}$ bis $+/-1.7*10^{308}$. Aus Platzgründen verzichte ich hier darauf, diese Zahlen auszuschreiben. Wir wissen bereits, dass bei 10^{-324} das Komma um 324 Stellen nach links wandert und bei 10^{308} um 308 Stellen nach rechts. Die Genauigkeit des `double`-Datentyps ist erheblich höher als die des `float`-Typs und liegt bei etwa 15 – 16 Stellen.

3.3.8 Der decimal-Typ

Mit den beiden Datentypen `float` und `double` decken Sie den größten Teil der Programmierarbeiten ab, bei denen Berechnungen durchgeführt werden. Bei präzisen wissenschaftlichen Aufgaben stoßen Sie aber bei Ihren Berechnungen mit diesen Datentypen schnell an die Grenzen der Genauigkeit. Es empfiehlt sich in solchen Fällen, den Datentyp `decimal` einzusetzen. Für diesen Datentyp werden 16 Byte im Arbeitsspeicher reserviert, also 128 Bit und damit doppelt soviel wie beim `double`-Typ. Man erhält eine sehr hohe Genauigkeit von 28-29 Stellen, dafür ist allerdings der Wertebereich kleiner als beim `double`-Typ, er liegt zwischen $1*10^{-28}$ bis $7.9*10^{28}$. Des Weiteren werden nur positive, keine negativen Werte abgedeckt.

Bei der Deklaration von `decimal`-Datentypen gilt das gleiche wie bei den `float`-Datentypen. Wie oben bereits erwähnt, werden echte Literale wie die Zahl 5.5 als `double` deklariert, daher muss ein Spezifizierungs-Suffix verwendet werden, im konkreten Fall `m` oder `M`. Im folgenden Beispiel fehlt dieses:

```
decimal decZahl = 5.5;
```

Daher gibt der Compiler folgende Fehlermeldung aus:

```
Literale des Typs "Double" können nicht implizit zum Typ 'decimal'
konvertiert werden. Verwenden Sie ein 'M'-Suffix, um eine Literale mit
diesem Typ zu erstellen.
```

Die richtige Deklaration von `decimal`-Datentypen lautet:

```
decimal decZahl = 5.5m;
```

Das folgende Listing soll Ihnen zeigen, dass die Verwendung von Gleitkomma-Datentypen bei kritischen Berechnungen bzw. sicherheitsrelevanten Systemen nicht immer so einfach zu handhaben ist. Die Verwendung von `float`, `double` und `decimal`-Datentypen muss daher gut durchdacht werden. Ich zeige hier eine einfache Berechnung, die je nach Wahl der Gleitkomma-Datentypen drei verschiedene Ergebnisse liefert. Hier wird von der Zahl `10` eine Zahl `9.9` subtrahiert, das mathematisch korrekte Ergebnis sollte damit *10.0 – 9.9 = 0.1* lauten.

Listing 3.4: Variablen_Genauigkeitsfehler.cs

```
01: using System;
02:
03: namespace Kap3
04: {
05:
06:    class CUps
07:    {
08:       static void Main(string[] args)
09:       {
10:          float fZahl1 = 10.0f;
11:          float fZahl2 = 9.9f;
12:          float fErgebnis = 0;
13:
14:          double dZahl1 = 10.0;
15:          double dZahl2 = 9.9;
16:          double dErgebnis = 0;
17:
18:          decimal decZahl1 = 10.0m;
19:          decimal decZahl2 = 9.9m;
20:          decimal decErgebnis = 0;
21:
22:          fErgebnis = fZahl1 - fZahl2;
23:          Console.WriteLine("{0}",fErgebnis);
24:
25:          dErgebnis = dZahl1 - dZahl2;
26:          Console.WriteLine("{0}",dErgebnis);
27:
28:          decErgebnis = decZahl1 - decZahl2;
29:          Console.WriteLine("{0}",decErgebnis);
30:
31:          Console.ReadLine();
```

```
32:         }
33:     }
34: }
```

```
0,1000004
0,0999999999999996
0,1
```

In den Zeilen 10 bis 12 werden drei Variablen `fZahl1`, `fZahl2` und `fErgebnis` vom Datentyp `float` deklariert und initialisiert. Gleitkommazahlen werden standardmäßig mit dem Datentyp `double` deklariert und heißen echte Literale. Um eine literale Konstante vom Typ `float` zu erzeugen, benötigen wir daher in den Zeilen 10 und 11 das Suffix `f`. In der Zeile 12 kann das Suffix `f` wegfallen, da bei Ganzzahlen (im Beispiel 0) die implizite Konvertierung die Arbeit für uns erledigt. Der Form halber sollte aber hier ebenfalls ein `f` geschrieben werden. Im Beispiel wurde es weggelassen, nur um zeigen, dass es auch in dieser Form akzeptiert wird.

In den Zeilen 14 bis 16 werden wieder drei Variablen, diesmal vom Typ `double` deklariert und initialisiert. Hier ist die Verwendung des Suffix nicht notwendig, da es sich hier um echte Literale und somit um den Datentyp `double` handelt. Wechseln Sie aber häufig in Ihrem Programm die Gleitkommazahltypen, ist auch hier die Verwendung des Suffix `d` (bzw. `D`) für den Datentyp `double` ratsam, um die Übersicht zu behalten.

In den Zeilen 18 bis 20 gilt das gleiche Prinzip wie in den Zeilen 10 bis 12. Hierbei handelt es sich aber um den Datentyp `decimal`. Das Suffix `m` der beiden Variablen `decZahl1` und `decZahl2` ist deshalb auch hier notwendig.

In den Zeilen 22, 25 und 28 werden die in den Variablen gespeicherten Werte voneinander subtrahiert. Es handelt sich im Prinzip immer um die gleiche Berechnung, aber mit verschiedenen Datentypen. Das jeweilige Ergebnis der drei Berechnungen unterscheidet sich jedoch je nach verwendetem Datentyp und der daraus resultierenden Genauigkeit.

Sie sehen, dass man die Verwendung von Datentypen gut überdenken muss. Bei systemkritischen Anwendungen greift man gerne auf die numerische Mathematik und hier speziell auf die Fehlerberechnung zurück. Hierbei unterscheidet man grob zwischen:

- verfälschten Eingabedaten
- Rundungsfehlern, die durch endliche Zahlendarstellungen verursacht werden
- Verfahrens- und Diskretisierungsfehlern
- echten Fehlern in numerischen Berechnungen

3.3.9 Der boolean-Typ

Der Datentyp `boolean` (auch als *Boolescher Datentyp* bezeichnet) kann einen von 2 Werten annehmen, `true` oder `false`. Der Datentyp hat damit viel mit einem Bit zu tun, das eben auch nur zwei Werte einnehmen kann: 0 und 1. Man könnte meinen, dass deshalb für eine Variable vom Typ `boolean` nur 1 Bit im Arbeitsspeicher belegt wird. Auf Grund der besonderen Verwaltung des Arbeitsspeichers wird aber tatsächlich ein komplettes Byte reserviert, also ebenso viel Speicher wie beim Datentyp `byte`.

Den `boolean`-Datentyp findet man in einem Programm meist für Entscheidungen bei Bedingungen. Der `boolean`-Typ arbeitet nach der positiven Logik: Die logische 1 bedeutet wahr – `true` – und die logische 0 falsch – `false`.

3.3.10 Ist das der richtige Typ?

Im Allgemeinen bleibt es dem Programmierer überlassen, für welchen Datentyp er sich entscheidet. Bei der Wahl des richtigen Datentyps sollten Sie aber an die Zukunft denken, d.h., das Programm sollte auch dann noch einwandfrei funktionieren, wenn z. B. die Datenmenge, die gespeichert wird, im Laufe der Zeit immer größer wird. Hier wäre es ungeschickt, Datentypen zu verwenden, die irgendwann nicht mehr ausreichen. (Ihnen ist sicherlich das Jahr-2000-Problem gut in Erinnerung, das zum wesentlichen Teil daraus entstanden ist, dass in den betroffenen Programmen zu wenig Speicher für die Jahreszahl reserviert wurde. Entsprechend kostspielig waren die nachträglich durchgeführten Anpassungsarbeiten.)

Heutzutage haben wir in der Regel ausreichend Arbeitsspeicher und entsprechend schnelle Hardware, sodass man im Zweifelsfall lieber den Datentyp mit einem entsprechend größeren Wertebereich wählen sollte. Jedoch sollte auf keinen Fall zu großzügig mit Datentypen umgegangen werden. Wenn z. B. absolut sicher ist, dass eine Schleife nur 10 Mal durchlaufen wird, wäre es übertrieben, für die betroffene Variable einen `int`-Datentyp zu verwenden. Es ist zu überlegen, ob stattdessen nicht ein `short`-Typ oder sogar ein `byte`-Typ ausreicht. Ein weiterer Aspekt ist die Performance (Leistungsfähigkeit und Schnelligkeit) eines Programms. Da die Performance nicht nur von dem Prozessor abhängig ist, auf dem das Programm läuft, sondern auch von dem verwendeten und vorhandenen Arbeitsspeicher in einem Computer, muss man sich auch hier Gedanken machen, welche Datentypen man verwendet. Der Datentyp `long` ist langsamer als der Datentyp `int`, da beim Ersteren eine höhere Anzahl an Informationen verarbeitet werden. Daraus folgt auch, dass der `long`-Datentyp mehr Arbeitsspeicher als der `int`-Datentyp benötigt.

3.4 Implizite Konvertierungen

Implizite Konvertierungen findet man in zahlreichen Situationen. Meist werden sie unbewusst vom Programmierer angewendet, ohne dass dieser groß weiter darüber nachdenkt. Hierbei kann es aber manchmal zu weniger schönen Nebeneffekten kommen. In der Tabelle 3.3 unten erhalten Sie einen Überblick über die wichtigsten impliziten Konvertierungen.

Von	In
`sbyte`	`short`, `int`, `long`, `float`, `double` oder `decimal`
`byte`	`short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` oder `decimal`
`short`	`int`, `long`, `float`, `double` oder `decimal`
`ushort`	`int`, `uint`, `long`, `ulong`, `float`, `double` oder `decimal`
`int`	`long`, `float`, `double` oder `decimal`
`uint`	`long`, `ulong`, `float`, `double` oder `decimal`
`long`	`float`, `double` oder `decimal`

Tabelle 3.3: Implizite Typkonvertierung

Von	In
`char`	`ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` oder `decimal`
`float`	`double`
`ulong`	`float`, `double` oder `decimal`

Tabelle 3.3: Implizite Typkonvertierung (Forts.)

Die implizite Konvertierung geschieht automatisch und kann zum Beispiel bei Gleitkommatypen zu einer Reduzierung der Genauigkeit bis hin zur Absolutwertdarstellung ohne Dezimalstellen führen.

3.5 Typumwandlung

Im strikten Gegensatz zur impliziten Konvertierung muss man bei der expliziten Konvertierung den Datentyp angeben, in den die Variable umgewandelt werden soll. Die Konvertierungsarten in der Programmiersprache C# hängen direkt von den Wertebereichen der verschiedenen Datentypen ab. Versucht man den Inhalt einer Variablen abzulegen, deren Datentyp einen größeren Wertebereich aufweist, als die Zielvariable, gibt der Compiler eine Fehlermeldung aus.

```
int iZahl = 0;
short sZahl = 0;

sZahl = iZahl;
```

Die dadurch erzeugte Fehlermeldung lautet:

```
Implizite Konvertierung des Typs 'int' zu 'short' nicht möglich.
```

Hier muss eine Typumwandlung der Variablen `iZahl` vorgenommen werden. Die Variable `iZahl` muss durch eine explizite Konvertierungsanweisung, man spricht dabei von *casten*, in den richtigen Datentyp umgewandelt werden. Dabei schreibt man den Datentyp, der erzeugt werden soll, in runden Klammern vor die Variable.

Die Anweisung für die explizite Konvertierung bzw. für die Typumwandlung lautet z. B. folgendermaßen:

```
sZahl = (short) iZahl;
```

3.5.1 Fehler bei der expliziten Konvertierung

Wir wissen, dass alle Datentypen eine bestimmte Größe im Arbeitsspeicher beanspruchen. Somit wird klar, dass durch die Typumwandlung von Datentypen größerer Wertebereiche in kleinere Wertebereiche Fehler auftreten können.

Der Datentyp `int` beansprucht 4 Byte im Arbeitsspeicher, der Datentyp `short` dagegen 2 Byte. Was geschieht nun, wenn man einen Wert einer Variablen vom Typ `int` in eine Variable vom Typ `short` umwandeln und ablegen möchte? Das folgende Listing demonstriert die damit verbundene Problematik.

Listing 3.5: Variablenkonvertierung.cs – Probleme

```
01: using System;
02
03: namespace Kap3
04: {
05
06:    class CProbleme
07:    {
08:       static void Main(string[] args)
09:       {
10:          int iZahl = 32768;
11:          short sZahl = 0;
12
13:          sZahl = (short)iZahl;
14
15:          Console.WriteLine("{0}",sZahl);
16:          Console.ReadLine();
17:       }
18:    }
19: }
```


```
-32768
```

Die Ausgabe ist vielleicht anders als erwartet, immerhin wurde der Wert gravierend verändert, aus 32768 wurde –32768. Hier muss man ein klein wenig tiefer in die interne Speicherverwaltung einsteigen. Die Variable `iZahl` ist vom Typ `int`.

Hierzu muss man wissen, dass bei vorzeichenbehafteten Datentypen wie `int` das erste Bit für das Vorzeichen verwendet wird, man spricht dabei von einem Vorzeichenbit.

Zunächst zum ursprünglichen `int`-Wert. Der Wert 32768 wird im Speicher durch 4 Byte repräsentiert, was so aussieht:

00000000 00000000 10000000 00000000

Der Wert ist positiv, das erste Bit – ganz links – lautet daher 0.

Der Datentyp `short` besitzt nur 2 Byte. Die ersten beiden Byte des 4-Byte-`short`-Werts werden daher bei der Typumwandlung ignoriert. Dies erscheint zunächst nicht weiter problematisch, sie enthalten scheinbar keine Information. Übrig bleibt die binäre Darstellung im Speicher

10000000 00000000

Da auch beim `short`-Datentyp das erste Bit für das Vorzeichen verwendet wird, ist aber nun ein negativer Wert entstanden, daher wurde aus dem ursprünglichen `int`-Wert der Wert -32768 vom Typ `short`. Man muss aber feststellen, dass es sich hierbei nicht direkt um einen Konvertierungsfehler handelt. Vielmehr ist die Ursache darin zu sehen, dass der gewählte Zieldatentyp zu klein ist, der höchste darstellbare `short`-Wert ist +32.767, es wurde aber versucht, den Wert +32.768 abzulegen. (Wenn Sie anstelle des `short`-Typs den `ushort`-Typ einsetzen, der auch nur zwei Byte aufweist, aber einen Wertebereich von 0 bis +65535, werden Sie feststellen, dass der richtige Wert +32.768 abgelegt wird.)

Für das Beispielprogramm gilt festzuhalten, dass das Ergebnis gravierend verändert wurde, ohne dass der Compiler eine Fehlermeldung ausgegeben hat. Bei der expliziten Konvertierung gilt im Allgemeinen eine prinzipiell einfache Regel. Wandelt man einen Wert in einen kleineren Typ um, werden alle Bits beginnend von rechts in den neuen Typ übernommen. Die anderen so genannten Überlaufbits werden verworfen.

3.5.2 Vermeiden von Konvertierungsfehlern

Die Fehlerbehandlung in C# wird durch die Ausnahmebehandlung (engl. *Exception*) abgedeckt und in Bonuskapitel 1 beschrieben. Da in der Programmierung Konvertierungsprobleme und Wertebereichsüberschreitungen zum Alltag gehören, gibt es in C# die Möglichkeit, solche Probleme zu überwachen. Oben haben wir gesehen, dass der Compiler keine Fehlermeldung ausgibt, wenn eine Bereichsüberschreitung durch die explizite Konvertierung ausgelöst wird. Wir müssen dem Compiler mitteilen, dass ein bestimmter Bereich in unserem Programm überwacht werden muss.

Konvertierungsfehler machen sich meist erst zur Laufzeit bemerkbar und sind für unerfahrene Programmierer nicht immer auf den ersten Blick zu erkennen. Wenn Sie auf der sicheren Seite sein möchten, können Sie alle Anweisungen, die Konvertierungen enthalten, in einen `checked`-Block setzen. Mit der Zeit werden Sie eine gewisse Routine für Konvertierungen bekommen und den `checked`-Block nur dort einsetzen, wo er benötigt wird. Das nächste Listing zeigt Ihnen den Einsatz eines `checked`-Blocks.

Listing 3.6: Variablen_checked.cs

```
01: using System;
02:
03: namespace Kap3
04: {
05:
06:    class CUmwandlung
07:    {
08:       static void Main(string[] args)
09:       {
10:          int iZahl = 32768;
11:          short sZahl = 0;
12:          checked
13:          {
14:             sZahl = (short)iZahl;
15:          }
16:          Console.WriteLine("{0}",sZahl);
17:          Console.ReadLine();
18:       }
19:    }
20: }
```

Um einen eventuellen Überlauf zu erkennen, wird die Anweisung, in der die Konvertierung ausgeführt wird, in einen `checked`-Block gekapselt. Der `checked`-Block überprüft während der Kompilierung den eingeschlossenen Block auf möglicherweise auftretende Fehler, die in der Laufzeit zu einem Programmabsturz führen könnten oder Ergebnisse verfälschen. Findet nun ein Überlauf statt, wird eine Ausnahme ausgelöst. Die Fehlermeldung, die Sie hier zur Laufzeit bekommen werden, lautet:

```
Unbehandelte Ausnahme:'System.OverflowException': Die arithmetische
Operation hat einen Überlauf verursacht.
```

An dieser Stelle stoppt das Programm und beendet sich. Um dies zu vermeiden, werden wir in einem späteren Kapitel noch ausführlich auf die Fehlerbehandlung eingehen. Wichtig zu wissen ist, dass die Überwachung von Konvertierungen nicht über Methodenaufrufe funktioniert.

3.6 Konstanten

Eine Konstante verhält sich prinzipiell genau wie eine Variable. Der einzige Unterschied ist, dass man eine Konstante nach ihrer Initialisierung nicht mehr verändern kann. In der Programmierung existieren zwei Arten von Konstanten: literale Konstanten (auch als echte, natürliche Konstanten bezeichnet) sowie symbolische Konstanten.

3.6.1 Literale Konstanten

Dieser Typ einer Konstanten wird durch seinen Wert repräsentiert. Literale Konstanten sind von Natur aus konstant und somit auch leicht als solche zu erkennen, da der Wert direkt im Quelltext in eingetippter Form zu finden ist. Beispiele für literale Konstanten sind:

```
int i = 1;
double d = 3.14;
char c = 'a';
string s = "Hallo .NET";
```

Im obigen Beispiel sehen Sie, dass neben Variablen auch literale und symbolische Konstanten einem bestimmtem Typ angehören. 1 ist ein Wert vom Typ `int` und 3.14 vom Typ `double`. Aber es sind nicht nur numerische Werte als Konstanten

erlaubt, sondern auch ein Zeichen oder eine Zeichenkette, die man dann als Stringkonstanten bezeichnet. So ist der einzelne Buchstabe `'a'` vom Typ `char` und `"Hallo .NET"` vom Typ `string`.

Wenn also eine Konstante genauso verwaltet wird wie eine Variable, wie kann man dann einer literalen Konstanten einen Datentyp zuweisen? Die Deklaration einer Variablen kennen wir bereits:

```
long iNumber = 50;
```

Aus dieser Deklaration ist ersichtlich, welchen Datentyp die Variable `iNumber` aufweist, nämlich `long`. `50` liegt im Wertebereich des Datentyps `long`. Für uns bedeutet dies, dass in der Variablen `iNumber` der Wert `50` abgelegt wird. Der Wert `50` repräsentiert standardmäßig den Typ `int`, was aber durch die automatische implizite Konvertierung nicht weiter problematisch ist, wie Sie gleich sehen werden.

Literalen Konstanten werden ohne die Verwendung eines Spezifizierungs-Suffix Standardtypen zugewiesen. Standardmäßig werden die Zahlen im Bereich von -2.147.483.648 bis 2.147.483.647 als `int` deklariert, also auch der Wert `50` aus dem obigen Beispiel. Gleitkommawerte werden als `double` deklariert. Sehen wir uns einfach einmal die einzelnen Standardtypen an:

Listing 3.7: Variablen_Typinfo.cs

```
01: using System;
02:
03: namespace Kap3
04: {
05:    class CGetType
06:    {
07:       public static void Main(string[] args)
08:       {
09:          Console.WriteLine("127-"+127.GetType());
10:          Console.WriteLine("255-"+255.GetType());
11:          Console.WriteLine("32767-"+32767.GetType());
12:          Console.WriteLine("32768-"+32768.GetType());
13:          Console.WriteLine("65535-"+65535.GetType());
14:          Console.WriteLine("2147483647-"+2147483647.GetType());
15:          Console.WriteLine("2147483648-"+2147483648.GetType());
16:          Console.WriteLine("9223372036854775807-
                              "+9223372036854775807.GetType());
17:          Console.WriteLine("1.33-"+1.33.GetType());
```

```
18:          Console.WriteLine("90000000.3333333333-
                               "+90000000.3333333333.GetType());
19:
20:          Console.ReadLine();
21:       }
22:    }
23: }
```

```
127-System.Int32
255-System.Int32
32767-System.Int32
32768-System.Int32
65535-System.Int32
2147483647-System.Int32
2147483648-System.UInt32
9223372036854775807-System.Int64
1.33-System.Double
90000000.3333333333-System.Double
```

In den Zeilen 7 bis 16 verwendet das Programm *Variablen_Typinfo.cs* die Methode `GetType()`. Diese Methode liefert hier den Datentyp der Konstanten zurück.

Das Beispiel zeigt, dass für alle Werte im Wertebereich des `int`-Datentyps der Datentyp `int` als Standardtyp gewählt wird, auch wenn Sie einen Wert angeben, für den z. B. der `byte`-Datentyp ausreichen würde. Nur für Werte außerhalb des `int`-Datentyps – ab 2147483648 – werden entsprechend andere Datentypen mit größerem Wertebereich verwendet.

In der Regel sind alle Wertetypen, deren Wertebereich kleiner oder gleich dem Datentyp der Zielvariablen ist, konform, sodass der Compiler eine implizite Konvertierung durchführen kann. Sie können somit einen Wert vom Typ `int` in einer Variablen vom Typ `long` speichern:

```
int iZ1 = 50;
long lZ2 = 0;
lZ2 = iZ1;
```

Möchte man aber von den Standardtypen bei den literalen Konstanten abweichen und den Datentyp selbst festlegen, muss man spezielle Spezifizierungs-Suffixe verwenden, die Sie bereits aus den vorgehenden Abschnitten kennen (z. B. `L` für den Datentyp `long`).

Wir wissen nun, dass der Wert `50` als literale Konstante vom Typ `int` ist. Möchte man aber, dass die literale Konstante `50` vom Typ `long` ist, muss die Anweisung mit dem Spezifizierer folgendermaßen aussehen:

```
int lZ1 = 50L;
long lZ2 = 0L;
lZ2 = lZ1;
```

Die dritte Zeile führt zu einer Fehlermeldung, obwohl der Wert `50` im Wertebereich des Datentyps `int` liegt. `50L` ist aber vom Typ `long`, weshalb die implizite Konvertierung nicht mehr funktioniert.

3.6.2 Symbolische Konstanten

Die Verwendung von literalen Konstanten funktioniert bei kleineren Programmen sehr gut. Für die Verwendung symbolischer Konstanten spricht aber, dass sie aussagekräftiger sind. Sie verstehen gerade in längeren Programmen besser, für was der Wert steht. Speziell, wenn ein und dieselben Werte mehrfach vorkommen, empfiehlt sich der Einsatz von symbolischen Konstanten: Möchten Sie den Wert einer Konstante nachträglich ändern, müssen Sie nicht mehrere Stellen im Quelltext bearbeiten, sondern nur die Stelle, an der die Konstante initialisiert wird.

Stellen Sie sich einmal vor, Sie schreiben ein sehr umfangreiches mathematisches Programm. Sie verwenden die Zahl `3.14` als Kreiszahl `PI` für Berechnungen und zufälligerweise taucht die Zahl `3.14` als cm-Wert für einen Seiteneinzug für den Ausdruck auf einem Drucker auf. Beim Testlauf Ihres Programms stellt man fest, dass die Rundungsfehler bei der Verwendung der Kreiszahl `PI`, also `3.14`, zu groß sind und bittet Sie darum, die Kreiszahl `PI` auf `3.1415` zu ändern. Ersetzen Sie alle Werte `3.14` mit der Suchen- und Ersetzen-Funktion des Editors, haben Sie auch den Wert für den Seiteneinzug mitgeändert.

Viel eleganter ist die Verwendung von symbolischen Konstanten. Zur Deklarierung einer Konstanten gehen Sie so vor wie bei Variablen, setzen aber das Schlüsselwort `const` vor die Anweisung. Vergessen Sie auch nicht, dass bei Konstanten die Initialisierung zusammen mit der Deklaration innerhalb ein und derselben Anweisung erfolgen muss:

```
const float PI = 3.14f;
const float PGLEFT = 3.14f;
```

3.7 Aufzählungstyp

Der Aufzählungstyp hilft Ihnen, sortierte Konstanten in einem eigenen Typ zusammenfassen zu können. Unter sortierten Konstanten versteht man Daten, die einer bestimmten alltäglichen Bezeichnung untergeordnet sind. Beispiele dafür sind die Monate eines Jahres oder die Tage einer Woche. Der Aufzählungstyp gehört zur Gruppe der symbolischen Konstanten, die durch einen Namen einen bestimmten Wert repräsentieren. Der Standardtyp der Aufzählung ist der `int`-Datentyp.

Die Deklaration des Aufzählungstyps wird durch das Schlüsselwort `enum` erzeugt. Die sortierten Konstanten werden anschließend in einer geschweiften Klammer zusammengefasst, wobei Sie die Konstanten durch Kommata trennen. Ganz am Ende steht wie bei jeder C#-Anweisung ein Semikolon:

```
enum Jahr{Jan,Feb,Mrz,Apr,Mai,Jun,Jul,Aug,Sep,Okt,Nov,Dez};
```

`enum` deklariert hier einen neuen Typ namens `Jahr`, der die Aufzählung beinhaltet.

`Jan`, `Feb`, ... sind symbolische Konstanten, die – wenn keine Zuweisung erfolgt – automatisch beginnend mit `0` inkrementiert werden. So besitzt die symbolische Konstante `Jan` den Wert `0`, `Feb` den Wert `1`, `Mrz` den Wert `2`, usw.

Möchte man aber mit einem bestimmtem Startwert einer Aufzählung beginnen, kann man dem ersten Element der Aufzählung einen Startwert zuweisen. Die nächsten Elemente besitzen dann um 1 inkrementierte Werte.

```
enum Jahr{Jan=1,Feb,Mrz,Apr,Mai,Jun,Jul,Aug,Sep,Okt,Nov,Dez};
```

Sind die Wertabstände der einzelnen Elemente aber größer als 1, ist es auch möglich, jedem Element einen Wert zuzuweisen.

```
enum TrackingNumber{OK=100,Unterwegs=200,Empfangen=300};
```

Das folgende Listing zeigt Ihnen die Verwendung des Aufzählungstyps.

Listing 3.8: Variablen_enum.cs

```
01: using System;
02:
03: namespace Kap3
04: {
05:    enum Woche
06:    {
07:       Montag = 1,
08:       Dienstag,
```

```
09:       Mittwoch,
10:       Donnerstag,
11:       Freitag,
12:       Samstag,
13:       Sonntag,
14:       Fehler = 500
15:    };
16:
17:    class CAufzaehlung
18:    {
19:       static void Main(string[] args)
20:       {
21:          Console.WriteLine(Woche.Montag + ":"
                               + Convert.ToInt32(Woche.Montag));
22:          Console.WriteLine(Woche.Dienstag + ":" +
                               Convert.ToInt32(Woche.Dienstag));
23:          Console.WriteLine(Woche.Mittwoch + ":" +
                               Convert.ToInt32(Woche.Mittwoch));
24:          Console.WriteLine(Woche.Donnerstag + ":" +
                               Convert.ToInt32(Woche.Donnerstag));
25:          Console.WriteLine(Woche.Freitag + ":" +
                               Convert.ToInt32(Woche.Freitag));
26:          Console.WriteLine(Woche.Samstag + ":" +
                               Convert.ToInt32(Woche.Samstag));
27:          Console.WriteLine(Woche.Sonntag + ":" +
                               Convert.ToInt32(Woche.Sonntag ));
28:          Console.WriteLine(Woche.Fehler + ":" +
                               Convert.ToInt32(Woche.Fehler));
29:
30:          Console.ReadLine();
31:       }
32:    }
33: }
```

```
Montag:1
Dienstag:2
Mittwoch:3
Donnerstag:4
Freitag:5
Samstag:6
Sonntag:7
Fehler:500
```

Die Zeilen 5 bis 15 deklarieren einen Aufzählungstyp mit dem Schlüsselwort `enum`. Da der Aufzählungstyp standardmäßig von 0 aus zu zählen beginnt, weise ich dem ersten Element `Montag` den Startwert 1 zu. Die symbolische Konstante `Dienstag` hat somit den Wert 2, `Mittwoch` den Wert 3 usw.

Eine Ausnahme wird in Zeile 14 gemacht. Hier weise ich der symbolischen Konstante `Fehler` den Wert 500 zu.

Die Zeilen 21 bis 28 geben mit der Methode `Console.WriteLine()` die einzelnen Elemente der Aufzählung als Namen aus sowie den dazugehörigen numerischen Wert, durch den die Elemente repräsentiert werden.

`Convert.ToInt32` wandelt den Wochentag in einen 32-Bit-Integerwert um. Eine weitere Möglichkeit wäre hier die Verwendung eines `(int)`-Präfixes, der die Umwandlung ebenfalls vornimmt und eine übersichtlichere Darstellung ermöglicht:

```
Console.WriteLine(Woche.Montag + ":" + (int)Woche.Montag);
```

3.8 Escape-Sequenzen

Zum Schluss noch ein kleiner Exkurs zu den so genannten Escape-Sequenzen, die in der Praxis sehr praktisch sind. In Zusammenhang mit der Formatierung von Strings werden Escape-Sequenzen noch einmal ausführlich behandelt (Kapitel 4.2) Escape-Sequenzen sind spezielle Zeichen, die Sie innerhalb von Stringkonstanten verwenden. Sie können damit Zeilenumbrüche realisieren oder Textausgaben allgemein formatieren. Außerdem helfen die Escape-Sequenzen bei Darstellungs- bzw. Formatierungsproblemen im Quellcode. Angenommen Sie möchten in einer Stringkonstante ein doppeltes Anführungszeichen darstellen. Im Quellcode bedeutet dies, dass der String hiermit beendet ist und es damit zu einer Fehlermeldung kommt:

```
string sName = "Hallo, ich heiße "Strasser""
```

Um hier Fehlermeldungen zu vermeiden, müssen Sie eine spezielle Escape-Sequenz verwenden. Der Compiler erkennt dieses spezielle Zeichen und setzt dann dafür ein doppeltes Anführungszeichen ein:

```
string sName = "Hallo, ich heiße \"Strasser\""
```

Die Syntax einer Escape-Sequenz sieht so aus: ein Backslash (\) gefolgt von einem Buchstaben, der die Formatierung beschreibt. Tabelle 3.4 listet einige wichtige Escape-Sequenzen auf:

Escape-Sequenz	Beschreibung
\n	Neue Zeile
\t	Tabulator
\b	Backspace
\"	Doppelte Anführungszeichen (")
\'	Einfaches Anführungszeichen (')
\\	Backslash (\)

Tabelle 3.4: Escape-Sequenzen

Das folgende Listing zeigt die Verwendung der Escape-Sequenz, die in eine neue Zeile umbricht.

```
using System;

namespace Kap3
{
   class CDemo
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Erste Zeile \n Zweite Zeile");

         Console.ReadLine();
      }
   }
}
```

```
Erste Zeile
Zweite Zeile
```

3.9 Anweisungen

Ein Programm wird durch verschiedene Anweisungen gesteuert. Anweisungen werten aber auch Ausdrücke aus (siehe Abschnitt 3.9.4), die das Programm weiterverarbeiten kann. Hierfür stehen ein Vielzahl von Operatoren – z. B. +, – und * – zur Verfügung.

Eine Anweisung, die man sehr oft in einem Programm sieht, ist die Zuweisung der Inhalte von zwei verschiedenen Variablen in eine Variable, die zu diesem Zweck durch einen Operator miteinander verknüpft werden:

```
a = b o c;
```

Das Zeichen `o` repräsentiert hier den Operator, der die beiden Variablen `b` und `c` miteinander verknüpft. Setzen Sie den entsprechenden Operator ein. Beispiele für Operatoren sind das Additionszeichen (+) und das Subtraktionszeichen (-). Das Ergebnis der Verknüpfung von `b` und `c` wird in der Variablen `a` gespeichert.

3.9.1 Whitespaces

Neben Anweisungen, Ausdrücken, Operatoren und Kommentaren kommen noch weitere Zeichen im Quelltext vor. Diese Zeichen nennt man *Whitespaces*. Sie werden in erster Linie zur sauberen, übersichtlichen Gestaltung des Quelltextes verwendet. Sicher haben Sie davon bereits ausgiebig Gebrauch gemacht, ohne sich dessen bewusst zu sein.

Immer, wenn Sie Zeilen einrücken und dazu die Tabulatortaste betätigen oder Zeilen mit der Eingabetaste umbrechen, werden Whitespaces eingefügt. In der Regel sind diese Zeichen nicht auf dem Bildschirm sichtbar, wobei sich die meisten Editoren auch so konfigurieren lassen, dass Whitespaces angezeigt werden (meist durch schwache Punkte oder Pfeile).

Zu den Whitespaces gehören das Leerzeichen und alle Zeichen, die ebenfalls Abstände erzeugen, vor allem der Tabulator und der Zeilenwechsel.

Da die Whitespaces in erster Linie zur Formatierung des Quelltextes dienen, werden sie zunächst einmal vom Compiler ignoriert. Hier sehen Sie einige Beispiele, wie Sie Anweisungen mit Hilfe von Whitespaces formatieren können.

```
a=b+c;
a = b + c;
a =   b + c;
a =
      b + c;
```

Alle Anweisungen bewirken das Gleiche. In der ersten Zeile wurden alle Zeichen nebeneinander geschrieben, was keinen besonders gut lesbaren Code ergibt. In den beiden darauffolgenden Zeilen wurden die Variablen und Operatoren durch Whitespaces optisch voneinander getrennt, was einen deutlich übersichtlicheren Code erzeugt. Im vierten Beispiel wurde die Verwendung von Whitespaces etwas übertrieben (zusätzlicher Zeilenumbruch), sodass die Lesbarkeit der Anweisung darunter leidet.

Sie sollten ausgiebig Gebrauch von Whitespaces machen, um Anweisungen, Blöcke und andere Elemente in Ihrem Quelltext einzurücken und anderweitig optisch zu trennen, damit der Code übersichtlich bleibt und gut lesbar ist.

Whitespaces werden jedoch nicht in allen Fällen vom Compiler ignoriert. Verwenden Sie Leerzeichen und Tabulatoren innerhalb eines Strings (Datentyp `string`) oder Characters (`char`), werden diese Zeichen Bestandteil der Daten, die im String bzw. Character gespeichert werden. Beispielsweise wird im String »Hello .NET« ein Leerzeichen zwischen den beiden Wörtern »Hello« und ».NET« gespeichert.

Die beiden folgenden Anweisungen haben das gleiche Resultat, wenngleich in der zweiten Anweisung ein Zeilenumbruch in den Methodename `Console.WriteLine` (diese Methode gibt Text auf die Konsole aus) eingefügt wurde. Die Zeilenumbrüche ändern ausschließlich die Formatierung des Quelltextes. Nur im String »Hallo .NET« wird das Leerzeichen ausgegeben:

```
Console.WriteLine
                ("Hallo .NET");

Console.
        WriteLine
                ("Hallo .NET");
```

Im Übrigen dürfen Sie innerhalb eines Strings oder Characters nur das Leerzeichen und den Tabulator verwenden:

```
Console.WriteLine("Hello→  .NET");
```

Im Beispiel werden die beiden Wörter durch den Tabulator – hier symbolisch durch einen Pfeil (→) dargestellt – auseinander gerückt, wobei im Vergleich zum normalen Leerzeichen ein deutlich größerer Horizontalabstand erzielt wird.

Zeilenvorschübe dürfen jedoch nicht innerhalb eines Strings oder Characters verwendet werden:

```
Console.WriteLine("Hallo
                          .NET");
```

Hier wurde innerhalb eines Strings in die nächste Zeile umbrochen, wodurch Sie folgende Fehlermeldung erhalten.

```
) wird erwartet
; wird erwartet
Zeilenvorschub in Konstante
```

Was ist aber, wenn Sie bei der Ausgabe bewusst einen Zeilenumbruch erzeugen möchten? In diesem Fall verwenden Sie das Escape-Zeichen `\n`, das ausführlich in Kapitel 4.2 beschrieben wird.

Sie sehen, dass nicht alle Arten von Whitespaces ignoriert werden. Mit der Zeit werden Sie aber ein Gespür dafür entwickeln, wie und wo Sie Whitespaces verwenden können und dürfen.

3.9.2 Leeranweisungen

Im Laufe der Zeit, wenn Sie andere C#-Programme zu Gesicht bekommen, wird Ihnen hier und da ein einsames Semikolon in einer Zeile auffallen. Solche Anweisungen nennt man Leeranweisungen. Man sieht solche Leeranweisungen zum Beispiel in `for`-Schleifen. Solche Anweisungen führen nichts aus. Der Sinn einer Leeranweisung wird in Kapitel 5.2.2 im Zusammenhang mit den Schleifen noch genauer erläutert. An dieser Stelle soll der Hinweis genügen, dass es sie gibt und dass sie in der Programmiersprache C# zulässig sind. Ein kleines Beispiel für eine zulässige Leeranweisung sehen Sie hier:

```
for (a = 0; a < 10; Console.WriteLine(a++))
;
```

3.9.3 Verbundanweisung und Blöcke

Im ersten Kapitel schilderte ich die Notwendigkeit von Blöcken, um die Übersichtlichkeit in einem Programm zu erhöhen. Solche Blöcke werden auch Verbundanweisungen genannt und fassen eine Gruppe von Anweisungen in einer geschweiften Klammer zusammen. Ein Block beginnt immer mit einer öffnenden geschweiften Klammer { und endet mit einer schließenden geschweiften Klammer }. Das folgende Beispiel zeigt Ihnen einen solchen Block.

```
{
   Console.WriteLine("{");
   Console.WriteLine("Hello");
   Console.WriteLine(".NET");
   Console.WriteLine("}");
}
```

Blöcke sieht man in der Programmierung in verschiedener Art und Weise:

```
{   Anweisung;
    Anweisung;
}

{
Anweisung;
Anweisung;
}

{EineEinzelneAnweisung;}
```

Alle diese Arten von Blöcken sind zulässig und werden in der Praxis verwendet. Die Unterschiede liegen ausschließlich in der Formatierung des Quelltextes, also was Einrückungen und Zeilenumbrüche angeht.

Ich möchte Ihrer Kreativität keinen Riegel vorschieben, aber gewöhnen Sie sich eine der oben gezeigten Blockdarstellungen an und erfinden Sie keine eigene. Diese dargestellten Blockanweisungen werden Sie in den meisten Programmen wiederfinden.

Sollten Sie irgendwann einmal ein fremdes Programm erweitern oder ändern, so führen Sie die Formatierung so weiter, wie sie der Entwickler des Programms eingeführt hat. Es gibt kaum etwas Schlimmeres in einem Quellcode als eine uneinheitliche Formatierung.

3.9.4 Ausdrücke

Ein Ausdruck kann ein Bestandteil einer Anweisung sein. Letztendlich sind Ausdrücke auch als Anweisungen zu sehen. In C# ist alles, was einen Wert zurückgibt und aus mindestens einem Element (z. B. Zahlenwert oder Variable) besteht, ein Ausdruck. Kurz gesagt: Alles, was einen Wert zurückliefert, ist ein Ausdruck, und da gibt es in C# praktisch unbegrenzt viele Möglichkeiten.

So liefern symbolische bzw. literale Konstanten und Variablen Ausdrücke zurück. Hier sehen Sie einige Beispiele für derartige Minimalausdrücke:

```
1.1        //liefert 1.1 - literale Konstante
PI.........//liefert 3.14 - symbolische Konstante (definiert)
x..........//liefert einen beliebigen Wert zurück - Variable
```

Das obere Beispiel zeigt Ausdrücke, die leicht erkennbar sind. Die folgende Anweisung besitzt vier Ausdrücke.

```
a = b + c;
```

Natürlich sind die drei Variablen auch Ausdrücke, die jeder für sich einen Wert beinhalten, den man – bevor die Anweisung ausgeführt wird – z. B. mit dem Debugger auswerten kann. Wo findet man aber den vierten Ausdruck? Die Anweisung selbst ist ein Ausdruck, da sie den Wert der Addition von `b` und `c` liefert. Lassen Sie sich nicht davon irritieren; das nächste Beispiel wird Ihnen Klarheit verschaffen.

Wenn `a = b + c` eine Anweisung ist und zugleich ein Ausdruck, da `a = b + c` einen Wert zurückliefert, dann müsste auch folgende Anweisung Gültigkeit haben.

```
z = a = b + c;
```

Stimmt, auch diese Anweisung ist zulässig, da `a = b + c` ebenfalls ein Ausdruck ist.

Beinhaltet `b` den ganzzahligen Wert 5 und `c` den ganzzahligen Wert 4, so ist klar, dass `a` den ganzzahligen Wert 9 repräsentiert, sofern alle Variablen mit dem Datentyp `int` deklariert worden sind. Der Ausdruck `a = b + c` beinhaltet auch den ganzzahligen Wert 9. Somit repräsentiert die Variable `z` ebenfalls den Wert 9.

`a` beinhaltet in diesem Beispiel die Addition der Variablen `b` und `c`. Der Wert von `a` ist immer die Summe der beiden Variablen `b` und `c` und somit auch der Wert, den die Variable `z` repräsentiert.

Hinsichtlich der Reihenfolge, in der Ausdrücke bewertet werden, spielt die Rangfolge »Punkt vor Strich« eine große Rolle; die Ihnen noch hinlänglich aus der Schulzeit bekannt sein dürfte. Man spricht dabei von der Rangfolge der Operatoren. Es werden also nicht alle Ausdrücke gleichrangig bewertet, sondern zuerst die Multiplikationen (Operator *) und Divisionen (/) ausgewertet und dann anschließend die Additionen (+) und Subtraktionen (-).

```
z = a * b + c;
```

Der Ausdruck `a * b` liefert hier einen Wert zurück, der anschließend mit dem Wert der Variablen `c` addiert wird, wodurch die gesamte Anweisung wieder zu einem Ausdruck wird.

3.10 Operatoren

Operatoren ermöglichen es, einen Ausdruck, eine Anweisung bzw. mehrere Ausdrücke und Anweisungen miteinander zu verknüpfen. In den vorangegangenen Abschnitten sind Ihnen bereits diverse Operatoren begegnet. Die Ausdrücke bzw. Anweisungen, die durch einen Operator miteinander verknüpft werden, nennt man Operanden. Das folgende Beispiel zeigt Ihnen eine einfache Verknüpfung anhand einer Addition.

```
a = b + c;
```

Das Pluszeichen (das Symbol für die Addition) ist hier der Operator; die beiden Variablen `b` und `c` (Elemente der Verknüpfung) stellen die Operanden dar. In C# gibt es eine Vielzahl solcher Operatoren:

- Zuweisungsoperator
- Mathematische Operatoren
- Restwertoperator
- Dekrement- und Inkrementoperatoren
- Zusammengesetzte Operatoren
- Vergleichsoperatoren
- Logische Operatoren
- Bedingungsoperator
- Kommaoperator

3.10.1 Zuweisungsoperator

Den Zuweisungsoperator haben wir schon des Öfteren verwendet. Das Symbol für den Zuweisungsoperator ist das Gleichheitszeichen =. Verwechseln Sie aber den Zuweisungsoperator in C# nicht mit dem Gleichheitszeichen ==, das in Formelsammlungen zu finden ist und zwei Variablen miteinander gleichsetzt oder vergleicht. Zum Vergleich von Werten gibt es in C# einen anderen Operator, den wir später kennen lernen werden.

Möchte man in C# einen Wert einer Variablen in eine andere Variable schreiben, so weist man die Variable einer anderen zu. Das sieht folgendermaßen aus:

```
a = b;
```

`a` gleich `b` bedeutet aber nicht, dass die beiden Variablen identisch im Wert sind, sondern dass der Wert der Variablen `b` in die Variable `a` geschrieben wird. Erst nach Durchführung dieser Zuweisung haben die beiden Variablen den gleichen Wert.

3.10.2 Mathematische Operatoren

Wir alle kennen die Grundrechnungsarten und deren Operatoren. Hierzu zählen:

- Addition
- Subtraktion
- Multiplikation
- Division
- (Vorzeichen)

Die allgemeine Schreibweise der mathematischen Verknüpfungen sieht folgendermaßen aus.

```
a = b o c;
```

Schreiben wir eine kleine Anwendung mit den Operatoren + und -. Das Programm fragt zwei numerische Werte vom Benutzer über die Konsole ab. Nach Eingabe dieser wird eine Berechnung durchgeführt und das Ergebnis ausgegeben.

Listing 3.9: Operatoren_Addition.cs

```
01: using System;
02:
03: namespace Kap3
04: {
05:    class CAddSub
06:    {
07:       static void Main(string[] args)
08:       {
09:          int iSumme = 0;
10:          int iDifferenz = 0;
11:          int iOperand1 = 0;
12:          int iOperand2 = 0;
13:
14:          iOperand1 = Convert.ToInt32(Console.ReadLine());
15:          iOperand2 = Convert.ToInt32(Console.ReadLine());
16:
17:          iSumme = iOperand1 + iOperand2;
18:          iDifferenz = iOperand1 - iOperand2;
19:
20:          Console.WriteLine("{0} + {1} = {2}",iOperand1,
                              iOperand2, iSumme);
21:
22:          Console.WriteLine("{0} - {1} = {2}",iOperand1,
                              iOperand2, iDifferenz);
23:
24:          Console.ReadLine();
25:       }
26:    }
27: }
```

```
3
5

3 + 5 = 8
3 - 5 = -2
```

In Zeile 7 beginnt der Hauptteil des Programms mit dem Funktionsaufruf `Main()`. In den Zeilen 9 bis 12 werden die Variablen, die in diesem Programm verwendet werden, deklariert und initialisiert. Die Zeilen 14 und 15 fragen zunächst die Werte vom Benutzer ab, die in die Konsole einge-

geben werden. Es sollen Ganzzahlberechnungen durchgeführt werden; da aber die Konsole nur mit dem Datentyp `string` arbeitet, müssen die eingegebenen Werte noch in das richtige Format umgewandelt werden. Die `Convert`-Klasse bietet verschiedene Möglichkeiten der Datentypkonvertierung. In diesem Programm wird der eingegebene Wert in einen 32-Bit-Integer-Wert umgewandelt. Die unterschiedlichen angebotenen Methoden richten sich nach dem Wertebereich und der Notwendigkeit:

- `ToInt16()` – 16 Bit Integer
- `ToInt32()` – 32 Bit Integer
- `ToInt64()` – 64 Bit Integer

In Zeile 17 werden die beiden Variablen `iOperand1` und `iOperand2` miteinander addiert und der Ausdruck `iOperand1 + iOperand2` der Variablen `iSumme` zugewiesen. In Zeile 18 werden die beiden Variablen `iOperand1` und `iOperand2` voneinander subtrahiert und der Ausdruck der Variablen `iDifferenz` zugewiesen. Die Zeilen 20 und 22 geben den Ausdruck der Verknüpfung und das Ergebnis dieser Verknüpfung auf die Konsole aus.

Die Addition und die Subtraktion sind recht trivial, abgesehen davon, dass man aus der Konsole nur den `string`-Datentyp erhält und die Werte für die weitere Verarbeitung umwandeln bzw. konvertieren muss.

Nicht ganz so einfach ist die Division. Hier muss man auf einiges achten. Kritisch sind vor allem die Division durch Null und das Verhalten von C# bei der Division von Ganzzahlen.

Listing 3.10: Operatoren_Division.cs

```
01: using System;
02:
03: namespace Kap3
04: {
05:    class CMulDiv
06:    {
07:       static void Main(string[] args)
08:       {
09:          int iErg1 = 0;
10:          int iErg2 = 0;
```

```
11:          int iOperand1 = 0;
12:          int iOperand2 = 0;
13:
14:          iOperand1 = Convert.ToInt32(Console.ReadLine());
15:          iOperand2 = Convert.ToInt32(Console.ReadLine());
16:
17:          iErg1 = iOperand1 * iOperand2;
18:          iErg2 = iOperand1 / iOperand2;
19:
20:          Console.WriteLine("{0} * {1} = {2}",iOperand1,
                               iOperand2, iErg1);
21:
22:          Console.WriteLine("{0} / {1} = {2}",iOperand1,
                               iOperand2, iErg2);
23:
24:          Console.ReadLine();
25:       }
26:    }
27: }
```

```
3
4

3 * 4 = 12
3 / 4 = 0
```

Bis zur Zeile 16 verhält sich das Programm wie das vorige. In der Zeile 17 wird eine Multiplikation der beiden Variablen `iOperand1` und `iOperand2` durchgeführt und das Ergebnis in der Variablen `iErg1` gespeichert. In Zeile 18 wird die Division durchgeführt. Diese Zeile betrachten wir nun genauer.

Alle drei Variablen dieser Anweisung sind vom Datentyp Integer. Wie wir bereits wissen, hat der Integertyp keine Nachkommastellen, er verarbeitet nur Ganzzahlen. Die Division von `3` durch `4` ergibt normalerweise – wenn mit Gleitkommazahlen gerechnet wird – `0.75`. Im Beispiel erhält man aber den Wert `0`. Da die Nachkommastellen vom Integertyp ignoriert werden, wird alles hinter dem Dezimalpunkt abgeschnitten. Wichtig zu wissen ist, dass keine Fehlermeldung vom Compiler oder vom Programm ausgegeben wird. So kann es leicht zu stark verfälschten Ergebnissen kommen, wenn man dies nicht beachtet. In diesem Fall

sollten Sie einen anderen Datentyp wie zum Beispiel `float` wählen. Kritisch sind ferner Divisionen durch Null; wenn Sie als zweiten Wert eine `0` eingeben, wird eine `DivideByZeroException` (wörtlich »Division durch Null«-Ausnahme) ausgelöst, da die Division durch Null mathematisch nicht definiert ist. Divisionen durch Null müssen entweder vermieden oder – wenn sie auftreten – entsprechend behandelt werden. Über die Ausnahmebehandlung erfahren Sie in Bonuskapitel 1 mehr.

3.10.3 Modulo, der Restwertoperator

Der Modulo-Operator liefert den Rest einer Division mit ganzen Zahlen zurück. So ergibt 5 modulus 4 den Wert 1, da bei der Division von 5 durch 4 der Wert 1 übrigbleibt. Der Modulo-Operator wird in C# durch das Zeichen `%` repräsentiert.

Es gibt eine Reihe von Anwendungsfällen, bei denen der Modulo-Operator eingesetzt werden kann. Angenommen, Sie möchten einen Datenbestand zeilenweise ausgeben und nach jeder 10. Zeile eine Zwischenlinie erzeugen, um eine bessere optische Trennung zu erreichen. In diesem Fall können Sie eine Berechnung von *Zeilennummer* modulus 10 durchführen. Wenn die Zeilennummer durch 10 geteilt wird, geht diese Division in jeder 10. Zeile ohne Restwert auf – Sie erhalten als Ergebnis 0 und können auf 0 abfragen. Immer wenn 0 zurückgeliefert wird, geben Sie eine Zwischenlinie aus. Ein Entwurf eines solchen Algorithmus könnte folgendermaßen aussehen:

```
AktuelleZeile % 10

...
95  % 10 -> 5
96  % 10 -> 6
97  % 10 -> 7
98  % 10 -> 8
99  % 10 -> 9
100 % 10 -> 0 -> Ausgabe einer Linie
...
```

Der Modulo-Operator arbeitet nur mit ganzen Zahlen und akzeptiert daher keine Gleitkommazahlen. Dies kann man sich z. B. bei der Division von Ganzzahlen zu Nutze machen, indem man neben dem eigentlichen Ergebnis der Division auch den Divisionsrest weiterverarbeitet. Das folgende Programm liest zwei ganze Zahlen ein und dividiert die erste Zahl durch die zweite Zahl. Der Rest der Division wird mit dem Modulo-Operator `%` berechnet.

```
01: using System;
02:
03: namespace Kap3
04: {
05:    class CDemo
06:    {
07:       static void Main(string[] args)
08:       {
09:          int dividend;
10:          int divisor;
11:
12:          Console.WriteLine("Geben Sie den Dividenden ein: ");
13:          dividend = Convert.ToInt32(Console.ReadLine());
14:
15:          Console.WriteLine
             ("Geben Sie den Divisor ein (keine 0): ");
16:          divisor = Convert.ToInt32(Console.ReadLine());
17:
18:          Console.WriteLine("{0} / {1}",dividend, divisor);
19:          Console.WriteLine("Ergebnis: {0}",dividend / divisor);
20:          Console.WriteLine("Der Rest der Division ist {0}",
             dividend % divisor);
21:
22:          Console.ReadLine();
23:       }
24:    }
25: }
Geben Sie den Dividenden ein: 3
Geben Sie den Divisor ein (keine 0): 2
3 / 2
Ergebnis: 1
Der Rest der Division ist 1
```

3.10.4 Inkrement- und Dekrement-Operatoren

In Schleifen oder bedingten Wiederholungen wird meistens eine Variable als Zählervariable, ein so genannter Counter, eingesetzt. Dieser hat die Aufgabe, seinen Wert bei jedem Durchlauf um 1 zu erhöhen oder zu verringern. Diesen Vorgang des Erhöhens nennt man inkrementieren und das Gegenteil, nämlich den Wert um eins zu verringern, dekrementieren. In C# finden sich spezielle Operatoren für solche Operationen.

So gibt es verschiedene Möglichkeiten, den Wert einer Variablen um eins zu erhöhen oder um eins zu verringern.

```
a = a + 1;
b = b - 1;

a++;
b--;

++a;
--b;
```

Die erste Art, den Wert einer Variablen um eins zu verändern, ist uns schon bekannt. Bei `a = a + 1` wird der aktuelle Wert der Variablen `a` um eins erhöht und wieder in die Variable `a` geschrieben. Die gleiche Vorgehensweise wird bei `b = b - 1` verwendet, nur dass hier der Wert um eins verringert wird.

Die anderen vier Beispiele `a++`, `b--`, `++a`, `--a` führen prinzipiell die gleichen Aktionen aus – inkrementieren und dekrementieren. Diese Darstellungsart bzw. die Art dieser Operatoren nennt man unäre Operatoren[4]. Unäre Operatoren wirken nur auf einen Operanden.

Was ist aber der Unterschied zwischen `a++` und `++a` bzw. `b--` und `--b`? Wenn sie als einzelne Anweisung in einer Zeile stehen, ist zunächst kein Unterschied feststellbar. Nur in Kombination mit einer erweiterten Anweisung ergibt sich ein Unterschied, dazu gleich mehr. Erst einmal führen wir einige Begriffe ein:

Steht der unäre Operator ++ bzw. `--` vor einer Variablen, so spricht man von Präfix-Inkrement-Operator bzw. Präfix-Dekrement-Operator.

```
++a;
--b;
```

Befindet sich der unäre Operator ++ bzw. `--` dagegen hinter einer Variablen, so spricht man von Postfix-Inkrement-Operator bzw. Postfix-Dekrement-Operator.

```
a++;
b--;
```

Die Präfix-Semantik in C# bedeutet, dass der Wert der Variablen zuerst inkrementiert bzw. dekrementiert wird und danach ausgelesen werden kann. Die Postfix-Semantik besagt, dass der Wert der Variablen nach dem Auslesen inkrementiert bzw. dekrementiert wird.

4 Unär kommt aus dem Lateinischen und bedeutet *einteilig*. Unäre Operatoren werden auch als monadische Operatoren bezeichnet.

```
int a = 1;
int b = ++a;

int c = 1;
int z = c++;
```

Das nächste Beispiel verdeutlicht die Post-Präfix-Semantik in der Programmiersprache C#.

Listing 3.11: Operatoren_Inkrement.cs

```
01: using System;
02:
03: namespace Kap3
04: {
05:    class CIncDec
06:    {
07:        static void Main(string[] args)
08:        {
09:           int a = 1;
10:           int b = 0;
11:           int c = 1;
12:           int z = 0;
13:
14:           b = ++a;
15:           z = c++;
16:
17:           Console.WriteLine("präfix: {0} ", b);
18:           Console.WriteLine("postfix:{0}", z);
19:
20:           b = --a;
21:           z = c--;
22:
23:           Console.WriteLine("präfix: {0} ", b);
24:           Console.WriteLine("postfix:{0}", z);
25:
26:           Console.ReadLine();
27:
28:        }
29:    }
30: }
```

```
präfix: 2
postfix:1
präfix: 1
postfix:2
```

In Zeile 14 wird die Präfix-Semantik auf die Variable `a` mit dem unären Operator ++ angewendet. Die Variable wurde in der Deklaration in Zeile 9 mit dem Wert `1` initialisiert. Der Inhalt der Variablen `a` wird inkrementiert, bevor er der Variable `b` zugewiesen wird. Die Variable `b` besitzt nun den Wert 2.

Anders in Zeile 15, wo die Postfix-Semantik auf die Variable `c` mit dem unären Operator ++ angewendet wird. Die Variable `c` beinhaltet ebenfalls den Wert `1`, der in der Deklaration in Zeile 11 initialisiert wurde. Jetzt sehen Sie den Unterschied zur Präfix-Semantik. Der Inhalt der Variablen `c` wird der Variablen `z` zugewiesen und danach inkrementiert. Das bedeutet, dass die Variable `z` den Wert `1` besitzt und nicht den Wert `2`.

Das gleiche Prinzip geschieht in den Zeilen 20 und 21, mit dem Unterschied, dass der unäre Operator `--` verwendet wird, sodass die Inhalte dekrementiert werden.

3.10.5 Zusammengesetzte Operatoren

Programmierer gelten meist als äußerst schreibfaul und vereinfachen häufig alles, was nur möglich ist. Das betrifft auch einige Operatoren, die man häufig verwendet. Es kommt z. B. häufig der Fall vor, dass Werte addiert werden und das Additionsergebnis einer Variablen zugewiesen wird. Das folgende Beispiel bildet das 5-er-Einmaleins in der Summendarstellung ab.

```
iErg = 5;          //1 * 5 = 5
Console.WriteLine(iErg);
iErg = iErg + 5;   //2 * 5 = 10
Console.WriteLine(iErg);
iErg = iErg + 5;   //3 * 5 = 15
Console.WriteLine(iErg);
...
```

Sie sehen, dass eine Menge an Schreibarbeit geleistet werden muss, um die ersten zehn Berechnungen im Quellcode abzubilden. In C# gibt es die Möglichkeit, die Schreibweise abzukürzen, falls mit einer Variablen eine Berechnung durchgeführt und das Ergebnis wieder derselben Variable zugewiesen wird. So kann man statt

```
iErg = iErg + 5;
```

auch

```
iErg += 5;
```

schreiben. Analysieren wir einmal folgendes Beispiel

```
01: using System;
02:
03: namespace Kap3
04: {
05:    class CDemo
06:    {
07:       static void Main(string[] args)
08:       {
09:          int a = 1;
10:          int b = 0;
11:          int c = 0;
12:          int z = 4;
13:
14:          b = a += 4;
15:          Console.WriteLine(b);
16:          c -= 4;
17:          Console.WriteLine(c);
18:          z *= 5;
19:          Console.WriteLine(z);
20:          z /= 5 + a ;
21:          Console.WriteLine(z);
22:          b  %= 4 ;
23:          Console.WriteLine(b);
24:
25:          Console.ReadLine();
26:
27:       }
28:    }
29: }
```

```
5
-4
20
2
1
```

In Zeile 14 wird der Wert von `a`, der zu Beginn des Programms den Wert `1` besitzt, um `4` erhöht und der Variablen `b` zugewiesen. Somit beinhalten die Variablen `a` und `b` den Wert `5`. In der Zeile 15 wird die Variable `b` ausgegeben. In Zeile 16 wird der Variablen `c` der Wert `4` abgezogen. Somit enthält die Variable `c` den Wert `-4`. Die Variable `z` beinhaltet den Wert `4`, der in der Zeile 18 mit `5` multipliziert wird. Die Variable `z` weist somit den Wert `20` auf. In der Zeile 20 wird eine Division mit der Variablen `z` durchgeführt. Zunächst einmal gilt ja Punkt vor Strich, die Division müsste also vor der Addition erfolgen. Schreibt man die Division der Zeile 20 in der ursprünglichen, anscheinend nur längeren Form, so kann man Folgendes erkennen.

```
z = z / 5 ;
```

Das wäre in unserem Fall `20 / 5 = 4`. Und `4 + a = 4 + 5 = 9`. Das ist aber nicht das Ergebnis, das in der Konsole zu sehen ist, denn ausgegeben wird `2`! Hier muss man Acht geben. `z /= 5 + a` ist nicht das Gleiche wie `z = z / 5 + a` ! Der Wert, der in der Variablen `a` steht (also `5`), wird im ersten Fall mit der Zahl `5` addiert und anschließend wird die Division vollzogen. Man erhält somit das Ergebnis von `20/ 10 = 2`.

In der Zeile 22 wird der Modulo-Operator eingesetzt. Der Wert der Variablen `b` ist `5` und der Rest von 5 / 4 ergibt 1. Dieser Wert wird auch wieder in die Variable `b` geschrieben.

3.10.6 Rangfolge und Klammern

Möchte man in C# Berechnungen durchführen, sollte man über die Rangfolge der Operatoren Bescheid wissen. In der Regel kann man auf das Wissen der Mathematikstunden in der Schule zurückgreifen.

```
a = 5 + 4 * 2;
```

Wie steht es nun in C# mit der Rangfolge der Operatoren? Wird zuerst der Term `5 + 4` berechnet und dann mit dem Faktor `2` multipliziert?

```
a = 9 * 2;
```

Oder wird die Regel, die wir aus der Schulzeit kennen – Punkt vor Strich – angewendet?

```
a = 5 + 8;
```

Wie bereits in den vorigen Abschnitten angedeutet, gilt bei den Grundrechenarten in C# die Regel Punkt vor Strich, das Ergebnis lautet damit `4*2+5=8+5=13`. Somit besitzt der Operator `*` eine höhere Priorität als der Operator `+`. In C# gibt es aber noch weitere Operatoren, sodass man mit der Regel »Punkt vor Strich« alleine nicht weiterkommt. Hierbei gilt eine genau festgelegte Rangfolge der Operatoren, nach der Berechnungen durchgeführt werden. Einige davon finden Sie in der folgenden Tabelle, wobei die Priorität von oben nach unten abnimmt.

unäre	`+ – ++ --`
multiplikative	`* / %`
additive	`+ -`

Tabelle 3.5: Priorität von Operatoren

Wie die Tabelle zeigt, stehen die unären Operatoren an erster Stelle. Hierzu zählen nicht nur die Inkrement- und Dekrementoperatoren, sondern auch die Vorzeichenoperatoren.

Die multiplikativen Operatoren kommen an zweiter Stelle. Lassen Sie sich nicht durch das Wort multiplikativ verwirren. Man kann nämlich jede Division auch als Multiplikation ausdrücken. So ist

```
z = a / b;
```

äquivalent zu

```
z = a * 1 / b;
```

Die additiven Operatoren kommen in der Rangfolge zum Schluss. Bei der Bezeichnung der Rangfolge gilt wieder das gleiche Prinzip wie oben. Man kann den Ausdruck

```
z = a - b;
```

auch als

```
z = a + -b;
```

schreiben. Betrachten wir noch einmal das Beispiel von oben.

```
z = a * 1 / b;
```

Dieser Ausdruck enthält mehrere Operatoren mit der gleichen Priorität. Hier gilt eine einfache Regel. Befinden sich wie in diesem Beispiel mehrere multiplikative Operatoren in einem Ausdruck, so wird das Ergebnis von links nach rechts ausgewertet. Das bedeutet, dass hier zuerst das Ergebnis der Multiplikation `a * 1` berechnet und anschließend das Ergebnis durch den Wert der Variablen `b` geteilt wird.

Die Analyse des folgenden Listings verdeutlicht die Auswirkungen der Rangfolge von Operatoren.

```
01: using System;
02:
03: namespace Kap3
04: {
05:    class CDemo
06:    {
07:       static void Main(string[] args)
08:       {
09:          int a = 10;
10:          int b = 5;
11:          int c = 2;
12:          int z = 0;
13:
14:          z = a * 1 / b;
15:          z = a + -b;
16:
17:          z = b / c * a;
18:          z = a * b % 20;
19:
20:          z = a++ * b - a;
21:          z = ++a * b - a;
22:
23:          Console.ReadLine();
24:        }
25:     }
26: }
```

Beginnen wir in der Zeile 14. Hier handelt es sich nur um multiplikative Operatoren, die die gleiche Priorität besitzen. Zuerst wird das Ergebnis von `a * 1` berechnet und anschließend durch den Wert der Variablen `b` dividiert. Der Inhalt der Variablen `z` ist somit der Wert `2`.

In Zeile 15 sehen wir einen additiven und einen unären Operator. Die unären Operatoren liegen in der Priorität über den additiven Operatoren und werden somit zuerst berechnet. Der unäre Vorzeichenoperator kehrt das Vorzeichen des Inhalts der Variablen `b` um. Steht hier eine negative Zahl, so wird der Wert positiv und umgekehrt. Anschließend wird der Wert der Variablen `b` mit der Variablen `a` addiert. Somit enthält der Inhalt der Variablen `z` den Wert `10 + -5 = 10 - 5 = 5`.

Die Zeilen 17 und 18 folgen den Regeln, die in der Zeile 14 beschrieben wurden. Erst einmal zu Zeile 17: Der Wert der Variablen `z` in dieser Zeile ist `20`. Sollte `b / c` nicht `2.5` ergeben? Hier ist jedoch zu beachten, dass mit einem Integer-Datentyp gearbeitet wird, sodass die Nachkommastellen ignoriert werden; daher lautet das Ergebnis nicht `2.5`, sondern `2` (was zum Ergebnis `2 * 10 = 20` für `z` führt). Zu Zeile 18: Das Ergebnis der Variable `z` ist der Wert `5 * 10 % 20 = 50 % 20 = 10`. Hier kann man sehr schön erkennen, dass Operatoren gleicher Priorität von links nach rechts verknüpft werden.

Die Zeilen 20 und 21 sind schon ein wenig delikater. Hier existieren unäre, multiplikative und additive Operatoren. Zunächst zur Zeile 20:

```
z = a++ * b - a;
```

Unäre Operatoren haben die höchste Priorität. `a++` inkrementiert in Zeile 20 den Wert, aber erst nachdem der Wert ausgelesen wurde. Es handelt sich hier um die Postfix-Notation. Der Wert der Variablen `a` ist in diesem Fall `10` (und nicht etwa `11`) und wird mit der Variablen `b` multipliziert (`10*5=50`). Erst nach dieser Operation wird `a` um eins erhöht.

Nun wird vom Ergebnis `50` der Wert `a` abgezogen, `a` hat aber jetzt den Wert `11`, weil `a` nach dem Auslesen inkrementiert wurde. Somit ergibt sich `50 - 11 = 39`. Nun zur Zeile 21:

```
z = ++a * b - a;
```

Hier besitzt die Variable `a` zunächst einmal weiterhin den Wert `11`. Da jetzt aber die Präfix-Notation verwendet wird (`++a`), wird der Wert bereits vor dem Auslesen inkrementiert, hat also dann den Wert `12`. Damit ergibt sich `12 * 5 - 12 = 60 - 12 = 48`.

Die Berechnung von Ausdrücken wird in C# mittels der Rangfolge von Operatoren gesteuert. Was ist aber, wenn man seine Berechnung selbst steuern möchte? Hier gibt es die Möglichkeit, indem man den bevorzugten Ausdruck in runde Klammern setzt. Klammern setzen die Reihenfolge, mit der Variablen verknüpft werden, außer Kraft, unabhängig davon, welcher Operator eingesetzt wird. Versuchen wir einmal im folgenden Beispiel, die Priorität zugunsten der Addition zu verändern.

```
z = a++ * b - a;
```

Wenn der Inhalt der Variablen `a` den Wert `10` besitzt und `b` den Wert `5`, so wissen wir schon aus dem vorhergehenden Listing, dass `z` den Wert `39` erhält. Wenn wir nun die Priorität zugunsten der Addition verändern wollen, müssen wir um den Ausdruck `b - a` eine Klammer setzen.

```
z = a++ * ( b - a );
```

Was denken Sie, wie das Ergebnis für die Variable `z` lautet? Greifen wir einmal auf unser Wissen zurück. C# wertet Ausdrücke in der Regel von links nach rechts aus. Die multiplikative Verknüpfung der Variablen `b` zu der Variablen `a++` wurde durch die Klammerstellung aufgehoben und somit der gesamte Ausdruck `b - a` multiplikativ verknüpft. In der Variablen `a` steht der Wert `10`, der mit dem Ausdruck `b - a` multipliziert wird. Der Ausdruck `b - a` enthält aber die Variable `a`, die natürlich zuvor um eins inkrementiert wurde. So ergibt sich der Ausdruck:

```
z = 10 * ( 5 - 11);
```

Bevor die Multiplikation stattfindet, wird der Klammerausdruck ausgewertet.

```
z = 10 * -6;
```

Der Inhalt der Variablen `z` ist somit der Wert `-60`.

3.10.7 Vergleichsoperatoren

Die Vergleichsoperatoren haben die Aufgabe, Ausdrücke miteinander zu vergleichen und damit Abfragen zu beantworten. Jede Abfrage hat aber eine gravierende Einschränkung, sie liefert als Ergebnis ausschließlich ja oder nein. So ist es nicht möglich, eine Frage zu stellen wie zum Beispiel »In welchem Verzeichnis liegt die Datei mit dem Namen *xyz.txt* ?«. Aber Sie können eine Routine schreiben, die folgende Frage beantworten kann – »Liegt die Datei mit dem Namen *xyz.txt* in diesem Verzeichnis?«.

Solche Ausdrücke können mit Hilfe von Vergleichsoperatoren bearbeitet werden. C# arbeitet mit der positiven Logik. So ergibt eine Frage, die mit ja beantwortet wird, als Ergebnis `true` (wahr); eine Frage, die als Ergebnis nein hat, dagegen `false` (falsch).

Die folgende Tabelle zeigt eine Aufstellung der Vergleichsoperatoren mit Beispielen dazu.

Operator		Beispiel (a = 10, b = 5)		Ergebnis
gleich	==	a gleich b?	a == b	0 , false
ungleich	!=	a ungleich b?	a != b	1 , true
größer	>	a größer b?	a > b	1 , true
kleiner	<	a kleiner b?	a < b	0 , false
größer gleich	>=	a größer oder gleich b?	a >= b	1 , true
kleiner gleich	<=	a kleiner oder gleich b?	a <= b	0 , false

Tabelle 3.6: Vergleichsoperatoren

Verwechseln Sie nicht den Zuweisungsoperator = mit dem Vergleichsoperator ==!

Wird einer Variablen ein Wert zugewiesen, um zum Beispiel Berechnungen durchführen zu können, dann verwenden Sie den Zuweisungsoperator. Hierbei wird der Inhalt der Variablen manipuliert. Der Inhalt der Variablen a besitzt hier den Wert 5:

```
a = 5;
```

Vergleicht man dagegen eine Variable mit einem Wert oder einer anderen Variable, um zum Beispiel danach Entscheidungen treffen zu können, verwendet man den Vergleichsoperator. Hierbei wird der Inhalt der Variable nicht manipuliert.

```
a == 10;
```

Der Inhalt der Variablen besitzt den Wert 5 und nicht den Wert 10. Hierbei liefert der gesamte Ausdruck false zurück.

if-Anweisung

Programme, die ausschließlich eine Anweisung nach der anderen ausführen, wären in der Praxis für die typischen Problemstellungen, die es zu lösen gilt, kaum zu brauchen – ihnen würde eine gewisse (künstliche) Intelligenz fehlen. Um aber von einem Programm Entscheidungen treffen lassen zu können, bedarf es speziel-

ler Anweisungen, die in der so genannten Ablaufsteuerung zusammengefasst werden. Die Ablaufsteuerung werden wir in Kapitel 3 noch genauer durcharbeiten.

Eine Verzweigung im Programmablauf besitzt mindestens eine Bedingung und aus dieser Bedingung eine resultierende Anweisung. Das Ergebnis oder der Wert einer Bedingung kann nur zwei Zustände enthalten: `true` oder `false`.

Der Zustand innerhalb der Ablaufsteuerung resultiert aus der positiven Logik. Ist der Wert der Bedingung wahr bzw. `true`, dann werden die Anweisungen ausgeführt, die im Verzweigungsblock stehen. Ist der Wert der Bedingung aber falsch bzw. `false`, werden die Anweisungen im Verzweigungsblock ignoriert.

Die einfachste Art und Weise, eine Entscheidung in einem Programm ausführen zu lassen, ist die `if`-Anweisung. Die Syntax dieser Verzweigung sieht folgendermaßen aus:

```
if(Bedingung)
   Anweisung;
```

oder

```
if(Bedingung)
{
   Anweisungen;
   ...;
   ...;
}
```

Stehen in einem Verzweigungsblock mehrere Anweisungen, spricht man von einer Verbundanweisung. Ist die Bedingung erfüllt, werden alle in der Verbundanweisung enthaltenen Anweisungen ausgeführt.

Die `if`-Anweisung wird nicht mit einem Semikolon beendet.

`if`-Anweisungen werden in der Programmierung grundsätzlich in Verbindung mit Ausdrücken eingesetzt, die miteinander verglichen werden. Das folgende Listing zeigt den Einsatz von `if`-Anweisungen.

Listing 3.12: Operatoren_Vergleich.cs

```
01: using System;
02:
03: namespace Kap3
```

```
04: {
05:    class CVergl
06:    {
07:       static void Main(string[] args)
08:       {
09:          int a = 0;
10:          int b = 0;
11:
12:          Console.WriteLine
             ("Geben Sie einen Integerwert ein. a= ");
13:          a = Convert.ToInt32(Console.ReadLine());
14:
15:          Console.WriteLine
             ("Geben Sie einen Integerwert ein. b= ");
16:          b = Convert.ToInt32(Console.ReadLine());
17:
18:          if(a == b)
19:          {
20:             Console.WriteLine("a({0}) == b({1})",a,b);
21:          }
22:
23:          if(a > b)
24:          {
25:             Console.WriteLine("a({0}) > b({1})",a,b);
26:          }
27:
28:          if(a < b)
29:          {
30:             Console.WriteLine("a({0}) < b({1})",a,b);
31:          }
32:
33:          Console.ReadLine();
34:       }
35:    }
36: }
```

```
Geben Sie einen Integerwert ein. a=
7
Geben Sie einen Integerwert ein. b=
9
a(7) < b(9)
```

Das Programm entscheidet, ob die erste Zahl, die Sie eingeben, größer, kleiner oder gleich der zweiten eingegebenen Zahl ist. Abhängig von den eingegebenen beiden Werten wird eine der drei Bedingungen erfüllt und die nachstehende Anweisung ausgeführt; das Ergebnis des Vergleichs wird dabei in die Konsole ausgegeben.

Im Einzelnen geschieht Folgendes: In Zeile 12 fordert das Programm den Benutzer auf, einen Integerwert einzugeben, der dann in Zeile 13 eingelesen und in einen 32-Bit-Integerwert umgewandelt wird. Diese Zahl wird der Variablen `a` zugewiesen. In Zeile 15 und 16 wird das gleiche Prinzip wie in den Zeilen 12 und 13 verwendet, wobei der Wert der zweiten Eingabe der Variablen `b` zugewiesen wird.

Besitzen die beiden eingegebenen Zahlen den gleichen Wert, wird die Bedingung in Zeile 18 erfüllt und die Anweisung in Zeile 20 ausgeführt. Ist der Wert der ersten Zahl, die in der Variablen `a` gespeichert wurde, größer als der Wert in der Variablen `b`, so wird die Bedingung in Zeile 23 erfüllt und die Anweisung in Zeile 25 abgearbeitet. Ist die erste eingegebene Zahl kleiner als die zweite, so wird die Bedingung in Zeile 28 erfüllt und die Anweisung in Zeile 30 ausgeführt.

else-Klausel

Wir kennen jetzt die `if`-Anweisung, die uns hilft, in einem Programm nach gewissen Kriterien Entscheidungen treffen zu lassen. Tritt eine Bedingung ein, so werden die Anweisungen im Anweisungsblock abgearbeitet.

Die Anweisungen in einer `else`-Klausel werden ausgeführt, wenn die Bedingung der `if`-Anweisung nicht erfüllt wird. Die `else`-Klausel wird auch als Alternativzweig bezeichnet. Zum einen erhöht die `else`-Klausel die Lesbarkeit einer `if`-Anweisung, zum anderen möchte und kann man in größeren Programmen nicht alle Bedingungsfälle berücksichtigen.

Die Syntax einer `if-else`-Anweisung sieht folgendermaßen aus:

```
if(Bedingung)
   Anweisung1;
else
   Anweisung2;
```

oder

```
if(Bedingung)
{
   Anweisungen;
   ...;
   ...;
}
else
{
   Anweisungen;
   ...;
   ...;
}
```

Das folgende Listing zeigt die Verwendung einer `if-else`-Anweisung. Der Ablauf des Programms ist so gehalten, dass bei einem Bestellwert unter 100 Euro die Portokosten von 5 Euro zum Bestellwert addiert werden. Ab einem Bestellwert von 100 Euro werden die Portokosten vom Versender übernommen. Außerdem wird ein Rabatt vom hundertsten Teil des Bestellwertes subtrahiert.

Listing 3.13: Operatoren_ifelse.cs – Verwendung von if-else

```
01: using System;
02:
03: namespace Kap3
04: {
05:    class CElse
06:    {
07:       static void Main(string[] args)
08:       {
09:          int iBstWert = 0;
10:          int iPorto = 5;
11:
12:          Console.WriteLine("Geben Sie den Bestellwert ein.");
13:          iBstWert = Convert.ToInt32(Console.ReadLine());
14:
15:          if(iBstWert < 100)
16:          {
17:             iBstWert += iPorto;
18:          }
19:          else
20:          {
```

```
21:             iBstWert -= (iBstWert / 100);
22:          }
23:
24:          Console.WriteLine("Gesamtsumme:{0}",iBstWert);
25:
26:          Console.ReadLine();
27:       }
28:    }
29: }
```

```
Geben Sie den Bestellwert ein.
70
Gesamtsumme:75

Geben Sie den Bestellwert ein.
110
Gesamtsumme:109
```

Die Zeilen 12 und 13 kennen wir bereits aus den vorhergehenden Beispielen. Hier wird der Bestellwert als Integer-Typ eingelesen und in das richtige Format umgewandelt. In Zeile 15 wird überprüft, ob der Bestellwert unter `100` Euro liegt. Ist das der Fall, also die Bedingung `iBstWert < 100` erfüllt, so werden die Portokosten von `5` Euro in Zeile 17 zum Bestellwert addiert. Zeile 17 verwendet für die Addition einen zusammengesetzten Operator.

Enthält die Variable `iBstWert` einen Wert größer gleich `100` Euro, so liefert die Bedingung in der `if`-Anweisung den Wert `false` und das Programm setzt den Ablauf in Zeile 19 fort. Alle Bestellungen, die den Bestellwert von `99` Euro übersteigen, werden in Zeile 21 abgearbeitet. In Zeile 24 wird die zu zahlende Gesamtsumme ausgegeben.

C#-Stilkonventionen und Verschachtelung

Es ist grundsätzlich jedem Programmierer freigestellt, wie er seinen Quellcode formatiert. Wenn Sie aber einen Quellcode schreiben, den Sie nach einigen Monaten noch lesen und verstehen möchten oder in einem Entwicklungsteam arbeiten, sollten Sie sich an einige Regeln halten. Die Schreibkonventionen, die in den vorangegangenen Beispielen in Verbindung mit der `if`-Anweisung bzw. der `if-else`-Anweisung gezeigt wurden, und deren Möglichkeiten der Verschachtelung

gelten bei allen Programmblöcken. Dazu gehören auch Schleifen wie `while`, `do while`, `for` usw., aber auch der Aufbau von Klassen und Namespaces.

Die Strukturierung des Quellcodes ist ein wichtiger Aspekt der Programmierung. Hier werden die drei häufigsten Arten gezeigt und beschrieben.

```
using System;

namespace NsX{
   class Class1{
      static void Main(string[] args){
         Anweisungen
      }
   }
}
```

Diese Art der Blockdarstellung ist platzsparend und trotzdem noch sehr gut lesbar. Manche Programmierer stört die abgesetzte geschweifte Klammer (wie wir sie bisher eingesetzt haben und die auch im folgenden Quellcode zu sehen ist); sie wählen deshalb diese Schreibweise. Man sollte hierfür aber schon eine gewisse Übung besitzen.

```
using System;

namespace NsX
{
   class Class1
   {
      static void Main(string[] args)
      {

      }
   }
}
```

In diesem Fall werden die Blöcke, in denen die weiteren Anweisungen stehen, abgesetzt. Der Nachteil dieser Struktur ist, dass sich die Zeilenanzahl deutlich erhöht. In dieser Darstellungsart ist aber die Zugehörigkeit der Anweisungen sehr gut erkennbar.

```
using System;

namespace NsX
   {
```

```
      class Class1
      {
         static void Main(string[] args)
         {

         }
      }
   }
```

Diese Schreibweise weicht von der vorigen nur durch die Einrückung der Blöcke ab. Der Nachteil dieser ist, dass sich die Zeilenlänge erhöht. Hat man sehr viele hierarchisch abgesetzte Blöcke im Quellcode, muss man häufig horizontal scrollen, um die komplette Zeile zu erfassen, was die Lesbarkeit beeinträchtigt.

Sollten Sie einmal das Problem haben, dass wichtige Teile des Quellcodes zu weit rechts stehen, sodass Sie laufend horizontal scrollen müssen, können Sie einen Block auch ausrücken – ganz an den linken Rand. Häufig wird dann im Block ein Kommentar mit dem Inhalt <- eingefügt, der symbolisiert, dass der folgende, ausgerückte Block eigentlich in der Hierarchie optisch hierhin gehört, entsprechend wird das Ende der Ausrückung erneut durch einen Kommentar <- gekennzeichnet. Das folgende Beispiel zeigt einen solchen Fall.

```
namespace NsX
   {
      class Class1
      {
         static void Main(string[] args)
         {
            if(Bedingung)
            {//<-
if(Bedingung)
{
   …
}
…
            }//<-

         }
      }
   }
```

Oftmals besitzt ein Anweisungsblock nur exakt eine Anweisung, die bei einer Bedingung ausgeführt wird. In diesem speziellen Fall ist es einem freigestellt, ob der Block mit geschweiften Klammern abgesetzt wird.

```
if(Bedingung)
{
   Anweisung1;
}
Anweisung2;
```

ist gleichbedeutend mit:

```
if(Bedingung)
   Anweisung1;
Anweisung2;
```

In beiden Fällen wird die `Anweisung1` ausgeführt, wenn die Bedingung erfüllt ist. `Anweisung2` wird immer ausgeführt, da sie nicht zum Anweisungsblock der Bedingung gehört.

Verzweigungen lassen sich auch beliebig verschachteln.

```
if(Bedingung1)
   Anweisung1;
else
   if(Bedingung2)
      Anweisung2;
Anweisung3;
```

Im obigen Beispiel enthält die `else`-Klausel eine weitere Verzweigung, die die `Anweisung2` ausführt, falls `Bedingung2` erfüllt ist. `Anweisung3` wird immer abgearbeitet.

3.10.8 Logische Operatoren

Bedingungen sind ebenfalls Ausdrücke, die man mit einzelnen Ausdrücken kombinieren kann. Logische Operatoren dienen dazu, mehrere Vergleiche in einer Bedingung auf einmal auszuführen.

Stellen Sie sich einmal ein Login-System vor, in dem man sich nur werktags einloggen kann. Hierbei müssen drei Kriterien erfüllt werden. »Ist der heutige Tag ein Werktag UND existiert der Benutzer im System UND stimmt das Passwort?« Um solche Ausdrücke auswerten zu können, stehen uns drei logische Operatoren zur Verfügung.

Operator		Beispiel
AND (und)	&&	Ausdruck1 && Ausdruck2
OR (oder)	||	Ausdruck1 || Ausdruck2
NOT (nicht)	!	!Ausdruck

Tabelle 3.7: Die logischen Operatoren

Die logische Anweisung AND vergleicht zwei Ausdrücke und wertet die Verknüpfung nach folgenden Regeln aus – Ausdruck1 && Ausdruck2

Ausdruck1	Ausdruck2	Ergebnis
true	true	true
true	false	false
false	true	false
false	false	false

Tabelle 3.8: Ergebnistabelle der AND-Verknüpfung

Bei einer Bedingung, in der zwei Ausdrücke mit einem logischen AND verknüpft werden, müssen beide Ausdrücke erfüllt sein, damit der Anweisungsblock ausgeführt wird.

Die logische Anweisung OR vergleicht zwei Ausdrücke und wertet die Verknüpfung nach folgenden Regeln aus – Ausdruck1 || Ausdruck2

Ausdruck1	Ausdruck2	Ergebnis
true	true	true
true	false	true
false	true	true
false	false	false

Tabelle 3.9: Ergebnistabelle der OR-Verknüpfung

Bei einer Bedingung, in der zwei Ausdrücke mit einem logischen `OR` verknüpft werden, muss mindestens einer der beiden Ausdrücke erfüllt werden, damit der Anweisungsblock ausgeführt wird.

Die logische Anweisung `NOT` negiert einen oder mehrere Ausdrücke.

Ausdruck	Ergebnis
`true`	`false`
`false`	`true`

Tabelle 3.10: Ergebnistabelle der NOT-Verknüpfung

Stellen Sie sich vor, Sie müssen ein Programm entwickeln, welches die verschiedenen Temperaturstufen von Fieber hinsichtlich der Einteilung leicht – stark – normal auswertet. Sie benötigen dazu logische Operatoren, die Ihnen die Zuteilung in diese Bereiche ermöglicht.

Listing 3.14: Operatoren_logische.cs – logische Operatoren

```
01: using System;
02:
03: namespace Kap3
04: {
05:    class CTemperatur
06:    {
07:       static void Main(string[] args)
08:       {
09:          int iTemp = 0;
10:
11:          Console.WriteLine("Körpertemperatur:");
12:          iTemp = Convert.ToInt32(Console.ReadLine());
13:
14:          if(iTemp < 36)
15:             Console.WriteLine("Unterkühlung");
16:          if(iTemp > 35 && iTemp < 38)
17:             Console.WriteLine("Normal");
18:          if(iTemp > 37 && iTemp < 40)
19:             Console.WriteLine("Leichtes Fieber");
20:          if(iTemp > 39)
```

```
21:             Console.WriteLine("Starkes Fieber");
22:
23:          Console.ReadLine();
24:       }
25:    }
26: }
```

```
Körpertemperatur:
34
Unterkühlung

Körpertemperatur:
36
Normal

Körpertemperatur:
38
Leichtes Fieber

Körpertemperatur:
40
Starkes Fieber
```

Das Prinzip der Zeilen 11 und 12 kennen wir bereits aus den vorherigen Beispielen. In der Integer-Variablen `iTemp` steht die eingegebene Temperatur, die von der Konsole eingelesen wurde.

Machen wir in der Zeile 16 weiter. Hier werden zwei Ausdrücke durch den logischen `&&`-Operator miteinander verknüpft, die einen bestimmten Temperaturbereich zwischen `36` und `37` Grad darstellen. Nur wenn beide Ausdrücke erfüllt werden und den Wert `true` zurückliefern, wird die Anweisung `Console.WriteLine("Normal")` ausgeführt. Das gleiche Prinzip wird auch in Zeile 18 angewandt.

Ein kleiner Tipp zum Schluss. Bevor Sie sich in tief verschachtelten `if`-Anweisungen verirren, verwenden Sie die logischen Operatoren.

3.10.9 Bedingungsoperator

Eine einfache `if-else`-Anweisung kann man auch mit dem Bedingungsoperator realisieren. Verwenden Sie diesen aber nur bei einfachen Ausdrücken, da ansonsten die Lesbarkeit des Quellcodes darunter leidet. Der Bedingungsoperator `? :` gehört zur Gruppe der ternären[5] Operatoren. Das Fragezeichen (`?`) leitet dabei den Ausdruck ein, der ausgeführt werden soll, wenn die Bedingung erfüllt ist. Der Doppelpunkt kennzeichnet den Beginn des Ausdrucks, der abgearbeitet werden soll, wenn die Bedingung nicht erfüllt ist. Die `if-else`-Anweisung

```
if(Bedingung1)
   Anweisung1;
else
   Anweisung2;
```

kann damit auch als

```
Bedingung1 ? Ausdruck1 : Ausdruck2;
```

geschrieben werden. Das folgende Beispiel zeigt die Verwendung des Bedingungsoperators und die programmtechnisch zugehörige `if-else`-Anweisung.

Listing 3.15: Operatoren_Bedingung.cs

```
01: using System;
02:
03: namespace Kap3
04: {
05:    class CBed
06:    {
07:       static void Main(string[] args)
08:       {
09:          int iZ1 = 0;
10:          int iZ2 = 0;
11:          int iZ3 = 0;
12:
13:          Console.WriteLine("Zahl 1 eingeben:");
14:          iZ1 = Convert.ToInt32(Console.ReadLine());
15:          Console.WriteLine("Zahl 2 eingeben:");
16:          iZ2 = Convert.ToInt32(Console.ReadLine());
17:
```

5 Ternär kommt aus dem Lateinischen und bedeutet *aus drei Teilen*. Ternäre Operatoren werden auch als triadische Operatoren bezeichnet.

```
18:          if(iZ1 > iZ2)
19:             iZ3 = iZ1;
20:          else
21:             iZ3 = iZ2;
22:          Console.WriteLine("Zahl 3: {0}",iZ3);
23:
24:          iZ3 = (iZ1 > iZ2) ? iZ1 : iZ2;
25:          Console.WriteLine("Zahl 3: {0}",iZ3);
26:
27:          Console.ReadLine();
28:       }
29:    }
30: }
```

```
Zahl 1 eingeben:
5
Zahl 2 eingeben:
8
Zahl 3: 8
Zahl 3: 8
```

Die Zeilen 13 bis 16 kennen wir bereits und die Zeilen 18 bis 22 dürften uns auch keine Probleme mehr bereiten. Interessant wird es in der Zeile 24. Hier wird der Bedingungsoperator verwendet. Wenn der Ausdruck `iZ1 > iZ2` true liefert, wird der Wert der Variablen `iZ1` in die Variable `iZ3` übertragen, andernfalls der Inhalt der Variablen `iZ2`.

3.11 Zusammenfassung

Die Verarbeitung und Manipulation von Daten sollten Sie immer mit einem gewissen Respekt behandeln. In diesem Kapitel haben Sie die wesentlichen Gefahren und Fehlerquellen kennen gelernt, die bei der Formatierung von Daten, bei Wertebereichsüberschreitungen und bei der Datentypkonvertierung existieren. Sie sollten sich immer Gedanken über den geeigneten Datentyp machen, bevor Sie diesen im Programm einsetzen. Vergessen Sie auch nicht, Variablen nach der Deklaration zu initialisieren. Machen Sie Gebrauch von symbolischen Konstanten, wo es Sinn ergibt.

Weiterhin haben Sie gelernt, was Ausdrücke, Anweisungen und Verbundanweisungen sind, wie man mit Operatoren umgeht und was eine `if`-Anweisung ist.

Wichtig beim Einsatz von Operatoren ist, dass man die Rangfolge beachtet, die Klammersetzung richtig verwendet und den Unterschied zwischen Operatoren und Vergleichsoperatoren kennt.

Die Formatierung des Quelltextes ist in der Regel frei wählbar. Man sollte sich aber hier an die Standardformatierung (vor allem was Einrückungen angeht) halten und gegebenenfalls bei Fremdprogrammen die vom Vorgänger verwendete Formatierung beibehalten.

3.12 Fragen und Antworten

F *Warum verwendet man bei Ganzzahlen nicht immer den Datentyp `long`, um Wertebereichsüberschreitungen weitgehend auszuschließen?*

A Bei den heutigen Computern ist das grundsätzlich kein Problem, diesen Lösungsansatz zu wählen. Der Grund liegt vielmehr darin, dass Ihr Programm nicht das einzige ist, das auf einem Rechner läuft. Wenn ein Wertebereich eines `short`-Typs ausreicht, sollte man nicht unnötigerweise einen `long`-Typ verwenden. Die Frage der Performance der Programme wird heute und in Zukunft eine große Rolle spielen. Stellen Sie sich einmal vor, dass ein Server 10 Programme gleichzeitig ausführt. Angenommen, jedes Programm verarbeitet durchschnittlich 500 Ganzzahlwerte; insgesamt sind dies 5.000 Ganzzahlwerte. Ein `short`-Typ belegt 2 Byte im Arbeitsspeicher, ein `long`-Typ dagegen vier Mal so viel, also 8 Byte. Bei Einsatz des `long`-Typs ergibt das einen Arbeitsspeicherverbrauch von immerhin 40.000 Byte, mit `short` werden lediglich 10.000 Byte belegt. Wenn man grundsätzlich Speicher verschenkt, addieren sich derartige Unterschiede. Nebenbei sind Datentypen mit kleinerem Wertebereich in der Regel effizienter; das Programm wird schneller ausgeführt.

F *Was sind hexadezimale Zahlen und welchen Vorteil bieten sie gegenüber dezimalen Zahlen?*

A Im täglichen Leben rechnen wir normalerweise mit dem Dezimalsystem. Das Dezimalsystem hat die Basis 10 (0,1,2,3,4,5,6,7,8,9,10,...). Das hexadezimale Zahlensystem hat eine Basis von 16. Erst ab der 16. Ziffer wird die Zahldarstellung zweistellig (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10,...). Für binäre Darstellungen und Berechnungen ist das hexadezimale Zahlensystem wie geschaffen, da die Basis 16 eine Zweierpotenz ist (2^4) und damit ausgezeichnet mit der binären Darstellung des Computers harmoniert. Mit einer hexadezimalen Ziffer lassen sich 16 Werte codieren, was 2^4 entspricht, also 4 Bit. Dies ist exakt ein halbes Byte. Daraus folgt, dass sich ein Byte durch 2 Hexziffern darstellen lässt, z. B. FF für binär 1111 1111. Im dezimalen System benötigen Sie dagegen bis zu 3 Ziffern, z. B. 255 für den Wert 1111 1111. Hexadezimale Werte und Binärwerte lassen sich einfach umrechnen; für jede Hexziffer kann der zugehörige 4-Bit-Wert eingesetzt werden. Hexadezimal F5A0 entspricht damit binär 1111 0101 1010 0000. Umrechnungen vom und in das Dezimalsystem sind dagegen viel aufwändiger.

F *Kann eine Bedingung auch einen Ausdruck enthalten?*

A Ja, jeder Ausdruck liefert einen Wert zurück, den Sie in einer Bedingung vergleichen können. So kann man z. B. `if((a = b - c) > 10)` schreiben.

F *Ist das Subtraktionszeichen (-) ein unärer oder binärer Operator?*

A Beides. Je nach Verwendung werden zwei Werte subtrahiert (dann fungiert das Subtraktionszeichen als binärer Operator) oder ein positiver Wert wird in einen negativen umgewandelt bzw. umgekehrt (dann wirkt das Subtraktionszeichen als unärer Operator).

3.13 Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

3.13.1 Quiz

1. Welche Variablennamen sind zulässig?

```
7Tage
_specialOrder
Postleitzahl
Eingang.Daten
Ausgang_Daten
decimal
Bez Name
Fläche
```

2. Welche Deklarationen sind zulässig?

```
int 123;
char Buchstabe;
long lZahl1, lZahl2;
int iZahl1, long iZahl2;
bool bWahl1 bWahl2;
short Arg = 7 + 7;
```

3. Wird folgendes Listing ausgeführt, oder wird eine Fehlermeldung ausgegeben?

```
static void Main()
{
   int iZahl = 0;
   string sAlter = "27 Jahre";

   iZahl = Convert.ToInt32(sAlter);
   Console.WriteLine(iZahl);
}
```

4. Wie bezeichnet man die folgende Anweisung und was bewirkt diese?

```
a = 1 + 1;
```

5. Was ist ein Ausdruck?

6. Welche Prioritäten haben diese Operatoren?

```
a >= oder >
b != oder ==
c == oder <
d * oder +
```

7. Welche Werte enthalten die Variablen `a` und `b`?

```
iCount = 10;

a = iCount++;
b = ++iCount;
```

3.13.2 Übungen

1. Mit welchen Datentypen würden Sie folgende Variablennamen deklarieren?

```
Vorname,
Alter,
Groesse,
Augenfarbe,
Gewicht,
Geschlecht
```

2. Deklarieren Sie eine Konstante, die den Wert von PI auf vier Dezimalstellen abbildet.
3. Deklarieren Sie einen Aufzählungstyp, der die Tageszeiten Morgen, Mittag, Abend und Mitternacht enthält.
4. Deklarieren und initialisieren Sie eine Variable `decimal` vom Typ `decimal` mit dem Wert 7,3.
5. Schreiben Sie ein Programm, welches zwei `int`-Werte von der Konsole einliest und die größere der beiden Zahlen ausgibt.
6. Wie lässt sich folgende Verschachtelung von `if`-Anweisungen vermeiden?

```
if(i > 100)
   if(i < 1000)
      Anweisung;
```

7. Ist das folgende Programm lauffähig?

```
using System;
namespace ConsoleApplication2
{class CU4{static void Main(string[] args){int a = 1;int b = 7;int c =
3;b = a++;
c = ++a;Console.WriteLine(b);Console.WriteLine(a);if(a = b)
Console.WriteLine("Hallo");Console.ReadLine();}}}
```

8. Formatieren Sie die Übung 7 derart, dass diese leichter lesbar wird, und beheben Sie den Fehler, wenn einer existiert.