

Objektorientierte Software-Entwicklung mit C#

Logische Operatoren und Vergleichsoperatoren,
Kontrollstrukturen und Schleifen, Methoden

Das Studienheft und seine Teile sind urheberrechtlich geschützt. Jede Nutzung in anderen als den gesetzlich zugelassenen Fällen ist nicht erlaubt und bedarf der vorherigen schriftlichen Zustimmung des Rechteinhabers. Dies gilt insbesondere für das öffentliche Zugänglichmachen via Internet, Vervielfältigungen und Weitergabe. Zulässig ist das Speichern (und Ausdrucken) des Studienheftes für persönliche Zwecke.

© Fernstudienzentrum Hamburg · Alle Rechte vorbehalten

Falls wir in unseren Studienheften auf Seiten im Internet verweisen/verlinken, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf Inhalt und Gestaltung haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

CSHP03D

Objektorientierte Software-Entwicklung mit C#

**Logische Operatoren und Vergleichsoperatoren,
Kontrollstrukturen und Schleifen, Methoden**

**Autor: Christoph Siebeck
Fachlektor: Torsten Schreiber**

Werden Personenbezeichnungen aus Gründen der besseren Lesbarkeit nur in der männlichen oder weiblichen Form verwendet, so schließt dies ausdrücklich alle anderen Geschlechtsidentitäten ein.

Falls wir in unseren Studienheften auf Seiten im Internet verweisen, haben wir diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten sollten.

Objektorientierte Software-Entwicklung mit C#

Logische Operatoren und Vergleichsoperatoren, Kontrollstrukturen und Schleifen, Methoden

Inhaltsverzeichnis

Einleitung	1
1 Logische Operatoren und Vergleichsoperatoren	3
1.1 Vergleichsoperatoren	3
1.2 Logische Operatoren	5
1.3 Kombination von logischen Operatoren mit Vergleichsoperatoren	10
Zusammenfassung	13
2 Kontrollstrukturen	15
2.1 Einfache Verzweigung mit if	15
2.2 Alternative Verzweigung mit if ... else	20
2.3 Geschachtelte Verzweigungen	22
2.4 Mehrfachauswahl mit switch ...case	26
Zusammenfassung	33
3 Schleifen	36
3.1 Kopfgesteuerte Schleife mit while	37
3.2 Fußgesteuerte Schleife mit do ... while	39
3.3 Zählschleife mit for	43
3.4 Die Anweisungen break und continue bei Schleifen	47
3.4.1 break	48
3.4.2 continue	52
3.5 Exkurs: Abfangen von ungültigen Eingaben	54
Zusammenfassung	55
4 Methoden	58
4.1 Methoden in C#	58
4.2 Methoden mit Rückgabewert	63
4.3 Methoden mit Argumenten	69
4.4 Tipps zum Arbeiten mit Methoden	73
Zusammenfassung	75

Schlussbetrachtung	77
---------------------------------	-----------

Anhang

A. Lösungen der Aufgaben zur Selbstüberprüfung	78
B. Glossar	83
C. Literaturverzeichnis	87
D. Abbildungsverzeichnis	88
E. Tabellenverzeichnis	89
F. Codeverzeichnis	90
G. Medienverzeichnis	91
H. Sachwortverzeichnis	92
I. Einsendeaufgabe	95

Einleitung

In diesem Studienheft stellen wir Ihnen zunächst noch einige weitere Operatoren für Vergleiche und logische Operationen vor. Anschließend erfahren Sie, wie Sie den Ablauf Ihrer Programme in Abhängigkeit von Bedingungen steuern können und wie Sie Anweisungen mit Schleifen wiederholen. Abschließend beschäftigen wir uns mit Methoden.

Im Einzelnen lernen Sie in diesem Studienheft:

- wie Sie Operatoren für Vergleiche verwenden,
- wie Sie logische Operatoren wie UND und NICHT einsetzen,
- wie Sie logische Operatoren und Vergleichsoperatoren kombinieren,
- wie Sie mit der `if`-Anweisung Befehle abhängig von Bedingungen ausführen lassen,
- wie Sie geschachtelte Verzweigungen erstellen,
- wie Sie Mehrfachauswahlen mit der Anweisung `switch ... case` erstellen,
- welche verschiedenen Schleifen C# unterstützt,
- wie Sie Anweisungen mit Schleifen wiederholen lassen,
- wie Sie die Verarbeitung von Schleifen mit `break` und `continue` beeinflussen,
- wie Sie in einer Schleife sicherstellen, dass Daten in der korrekten Form eingelesen werden,
- wie Sie mit Methoden Quelltext einmal erstellen und dann beliebig oft verwenden,
- wie Sie Rückgabewerte aus Methoden verarbeiten und
- wie Sie Werte an Methoden übergeben.

Beginnen wir mit den logischen Operatoren und den Vergleichsoperatoren.

Christoph Siebeck

1 Logische Operatoren und Vergleichsoperatoren

*In diesem Kapitel werden wir uns mit zwei weiteren Operatortypen beschäftigen: den **Vergleichsoperatoren** und den **logischen Operatoren**.*

1.1 Vergleichsoperatoren

Vergleichsoperatoren werden – wie der Name schon sagt – für den Vergleich von Daten benutzt. Sie können zum Beispiel zwei Zahlen miteinander vergleichen, eine Zahl mit dem Wert einer Variablen oder auch die Werte in zwei Variablen.

C# unterstützt folgende Vergleichsoperatoren:

Tab. 1.1: Vergleichsoperatoren von C#

Operator	Bedeutung
==	Es wird überprüft, ob die beiden Operanden gleich sind.
!=	Es wird überprüft, ob die beiden Operanden ungleich sind.
<	Es wird überprüft, ob der linke Operand kleiner ist als der rechte Operand.
>	Es wird überprüft, ob der linke Operand größer ist als der rechte Operand.
<=	Es wird überprüft, ob der linke Operand kleiner oder gleich dem rechten Operanden ist.
>=	Es wird überprüft, ob der linke Operand größer oder gleich dem rechten Operanden ist.

Bitte beachten Sie:

Der Vergleichsoperator `==` besteht aus **zwei** Zeichen. Achten Sie sorgfältig darauf, dass Sie ihn nicht mit dem Zuweisungsoperator `=` verwechseln.



Achten Sie auch darauf, Operatoren, die aus zwei Zeichen bestehen, zusammenzuschreiben. Beim Operator `<=` darf zum Beispiel zwischen den beiden Zeichen kein Leerzeichen stehen.

Tipp:

Wenn Sie sich den Unterschied zwischen den Operatoren `<` und `>` nicht merken können, gibt es eine einfache „Eselsbrücke“: Die Spitze zeigt immer zu dem kleineren der beiden Werte beziehungsweise zu dem Wert, der auf kleiner als überprüft werden soll.

Das Ergebnis eines Vergleichs ist entweder **wahr** oder **falsch** beziehungsweise `true` oder `false`.

Einige Beispiele für den Einsatz von Vergleichsoperatoren finden Sie im folgenden Code.

```

/* #####
Beispiele für Vergleichsoperatoren
##### */

using System;

namespace Cshp03d_01_01
{
    class Program
    {
        static void Main(string[] args)
        {
            int zahl1, zahl2;
            zahl1 = 1;
            zahl2 = 0;

            //ein direkter Vergleich von zwei Zahlen
            Console.WriteLine("5 < 10 \t{0}", 5 < 10);

            //ein Vergleich einer Zahl mit einer Variablen
            Console.WriteLine("5 > {0} \t{1}", zahl1, 5 > zahl1);

            //Vergleiche von zwei Variablen
            //bitte jeweils in einer Zeile eingeben
            Console.WriteLine("{0} == {1} \t{2}", zahl1, zahl2,
                zahl1 == zahl2);
            Console.WriteLine("{0} != {1} \t{2}", zahl1, zahl2,
                zahl1 != zahl2);
        }
    }
}

```

Code 1.1: Beispiele für Vergleichsoperatoren**Zur Auffrischung:**

Mit der Anweisung `using System;` wird der Namensraum `System` für den gesamten Code vereinbart. Dadurch ersparen Sie sich die ständige Eingabe von `System.` vor vielen Anweisungen.

Die Anweisung `namespace Cshp03d_01_01` vereinbart einen eigenen Namensraum für das Programm.

Das Programm sollte die folgende Ausgabe erzeugen:

```

5 < 10 True
5 > 1 True
1 == 0 False
1 != 0 True

```

Schauen wir uns an, wie diese Ausgaben zustande kommen.

Zuerst vergleichen wir mit dem Ausdruck `5 < 10` (5 kleiner als 10) zwei Zahlen. Das Ergebnis ist wahr. Deshalb wird `True` ausgegeben.

Danach vergleichen wir die Zahl 5 mit dem Wert der Variablen `zahl1`. Der Ausdruck ergibt `5 > 1` (5 größer als 1) und ist ebenfalls wahr.

Anschließend vergleichen wir die beiden Variablen `zahl1` und `zahl2` miteinander. Der Ausdruck `zahl1 == zahl2` (`zahl1` gleich `zahl2`) liefert `1 == 0` und ist falsch. Deshalb wird hier `False` ausgegeben. Der Ausdruck `zahl1 != zahl2` (`zahl1` ungleich `zahl2`) dagegen liefert `1 != 0` und ist wahr. Also wird `True` ausgegeben.

Probieren Sie einmal aus, was geschieht, wenn Sie den Operator `==` in der vorletzten Anweisung durch den Zuweisungsoperator `=` ersetzen. Sie werden sehen, der Compiler akzeptiert die Anweisung ohne Murren. Als Ergebnis wird dann allerdings

```
1 == 0 0
```

ausgegeben. Denn jetzt liefert der Ausdruck `zahl1 = zahl2` ja nicht mehr `true` oder `false`, sondern das Ergebnis der Zuweisung – also den numerischen Wert 0. Entsprechend wird dann in der letzten Anweisung auch der Wert 0 als Ergebnis der Zuweisung ausgegeben.

Noch einmal, weil es schnell für Fehler sorgen kann:

Achten Sie vor allem bei Ihren ersten Versuchen sehr sorgfältig darauf, dass Sie für Vergleiche den Operator `==` benutzen und nicht versehentlich den Operator `=`. Denn auch eine Zuweisung liefert ein Ergebnis – nämlich den Wert, der zugewiesen wurde.



So viel zu den Vergleichsoperatoren. Kommen wir nun zu den logischen Operatoren.

1.2 Logische Operatoren

Mit logischen Operatoren können Sie Ausdrücke – zum Beispiel Ergebnisse von Vergleichen – miteinander verknüpfen. Die logischen Operatoren liefern genau wie die Vergleichsoperatoren entweder wahr bzw. `true` oder falsch bzw. `false`.

Sie brauchen logische Operatoren, wenn Sie Anweisungen wie „Wenn es **nicht** regnet, dann gehe ich spazieren“ in Ihren Programmen umsetzen wollen. Wie das geht, werden Sie noch in diesem Heft lernen. Hier wollen wir Ihnen die drei wichtigsten logischen Operatoren daher zunächst einmal nur vorstellen. Sie finden sie in der folgenden Tabelle:

Tab. 1.2: Logische Operatoren von C#

Operator	Bedeutung
!	Logische Verneinung (NICHT, NOT)
&&	Logisches Und (UND, AND)
	Logisches Oder (ODER, OR)

Hinweis:

Das Zeichen `|` erhalten Sie mit der Tastenkombination `Alt Gr` + `⇧|`.

**Bitte beachten Sie:**

Die logischen Operatoren `&&` und `||` bestehen aus **zwei** Zeichen und nicht aus einem. Mit den Operatoren `&` und `|` führen Sie bitweise Verknüpfungen durch beziehungsweise Sie verknüpfen Bedingungen ohne optimierte Auswertung. Damit wollen wir uns hier aber nicht weiter beschäftigen.

Schauen wir uns die logischen Operatoren der Reihe nach an. Beginnen wir mit der **logischen Verneinung** – dem logischen NICHT.

NICHT ist der einfachste logische Operator. Er dreht das Ergebnis einer logischen Aussage um – macht also aus wahr falsch und aus falsch wahr. Im praktischen Einsatz sehen Sie den Operator `!` im folgenden Code. Das Programm verwendet die gleichen Ausdrücke wie der vorige Code und kehrt die Wahrheitswerte über den Operator `!` um:

```
/* #####
Logisches NICHT
##### */

using System;

namespace Cshp03d_01_02
{
    class Program
    {
        static void Main(string[] args)
        {
            int zahl1, zahl2;
            zahl1 = 1;
            zahl2 = 0;

            //ein direkter Vergleich von zwei Zahlen
            Console.WriteLine("!(5 < 10) \t{0}", !(5 < 10));

            //bitte jeweils in einer Zeile eingeben
            //ein Vergleich einer Zahl mit einer Variablen
            Console.WriteLine("!(5 > {0}) \t{1}", zahl1, !(5 > zahl1));
            //Vergleiche von zwei Variablen
            Console.WriteLine("!({0} == {1}) \t{2}", zahl1,
                zahl2, !(zahl1 == zahl2));
            Console.WriteLine("!({0} != {1}) \t{2}", zahl1,
                zahl2, !(zahl1 != zahl2));
        }
    }
}
```

Code 1.2: Logisches NICHT

Die Ausgabe sollte jetzt so aussehen:

```
!(5 < 10) False
!(5 > 1) False
!(1 == 0) True
!(1 != 0) False
```

Die Wahrheitswerte werden also umgedreht. Der Ausdruck `5 < 10` (5 kleiner als 10) ist eigentlich wahr und liefert das Ergebnis `True`. Durch den Operator `!` wird der Wahrheitswert umgedreht und zu `False`.

Bitte beachten Sie:

Damit der vorige Code übersetzt wird, müssen Sie die gesamten Ausdrücke über den Operator `!` umdrehen. Der Operator `!` muss also jeweils vor der Klammer `(` stehen. Andernfalls würden Sie versuchen, den Wahrheitswert einer Zahl umzudrehen – und das ist in C# nicht möglich.

Achten Sie deshalb beim Einsatz von Operatoren sehr sorgfältig auf die Klammern.



So viel zur logischen Verneinung.

Das **logische ODER** `||` verknüpft zwei Aussagen und liefert immer dann den Wert wahr zurück, wenn mindestens eine der Aussagen wahr ist. Umgangssprachlich lässt sich das logische ODER zum Beispiel so darstellen:

„Wenn die Sonne scheint **oder** wenn es warm ist, dann kann ich ein T-Shirt anziehen.“

Um ein T-Shirt anzuziehen, reicht es, wenn **entweder** die Sonne scheint **oder** wenn es warm ist.

Es gibt nur einen einzigen Fall, bei dem das logische ODER den Wert falsch zurückliefert – nämlich dann, wenn beide Aussagen den Wert falsch haben.

Die verschiedenen möglichen Kombinationen von zwei Wahrheitswerten durch das logische ODER finden Sie in der folgenden **Wahrheitstabelle** zusammengefasst.

Tab. 1.3: Wahrheitstabelle für das logische ODER

A	B	A B
wahr	wahr	wahr
wahr	falsch	wahr
falsch	wahr	wahr
falsch	falsch	falsch

Eine ähnliche Tabelle erzeugt auch der folgende Code. Es verknüpft die Werte `true` und `false` mehrfach logisch ODER.

```
/* #####
Logisches ODER
##### */

using System;
```

```

namespace Cshp03d_01_03
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("A \t B \t\t A || B");
            Console.WriteLine("wahr \t wahr \t\t {0}", (true || true));
            //bitte jeweils in einer Zeile eingeben
            Console.WriteLine("wahr \t falsch \t {0}", (true ||
            false));
            Console.WriteLine("falsch \t wahr \t\t {0}", (false ||
            true));
            Console.WriteLine("falsch \t falsch \t {0}", (false ||
            false));
        }
    }
}

```

Code 1.3: Logisches ODER**Noch einmal zur Erinnerung:**

Die logischen Werte heißen `true` und `false` mit kleinem Anfangsbuchstaben. Ausgegeben werden aber immer `True` beziehungsweise `False` mit großem Anfangsbuchstaben.

Jetzt bleibt uns noch der letzte logische Operator – nämlich `&&` für das **logische UND**.

Dieser Operator liefert nur dann wahr, wenn auch die beiden verbundenen Ausdrücke wahr sind. Für jeden anderen Fall wird falsch geliefert. Umgangssprachlich lässt sich das logische UND so darstellen:

„Wenn der Stecker in der Steckdose steckt **und** wenn das Radio eingeschaltet ist, höre ich Musik.“

Hier müssen also beide Ausdrücke wahr sein. Wenn nur der Stecker in der Steckdose steckt, das Radio aber nicht eingeschaltet ist, hören Sie keine Musik. Und wenn nur das Radio eingeschaltet ist, aber der Stecker nicht eingesteckt ist, hören Sie ebenfalls keine Musik.

Die Wahrheitstabelle für das logische UND sieht also so aus:

Tab. 1.4: Wahrheitstabelle für das logische UND

A	B	A && B
wahr	wahr	wahr
wahr	falsch	falsch
falsch	wahr	falsch
falsch	falsch	falsch

Das Beispielprogramm sieht fast genauso aus wie bei dem logischen ODER:

```
/* #####
Logisches UND
##### */

using System;

namespace Cshp03d_01_04
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("A \t B \t\t A && B");
            //bitte jeweils in einer Zeile eingeben
            Console.WriteLine("wahr \t wahr \t\t {0}", (true &&
            true));
            Console.WriteLine("wahr \t falsch \t {0}", (true &&
            false));
            Console.WriteLine("falsch \t wahr \t\t {0}", (false &&
            true));
            Console.WriteLine("falsch \t falsch \t {0}", (false &&
            false));
        }
    }
}
```

Code 1.4: Logisches UND

Bitte beachten Sie:

Die Auswertung von logischen Verknüpfungen über die Operatoren `&&` und `||` wird nicht immer vollständig ausgeführt, sondern intern optimiert. Bei einer logischen UND-Verknüpfung wird die Auswertung zum Beispiel abgebrochen, sobald der erste Ausdruck falsch ergibt. Denn dann wird ja auch zwangsläufig das Ergebnis der gesamten Verknüpfung falsch, da bei einer logischen UND-Verknüpfung alle Ausdrücke wahr sein müssen, damit das Gesamtergebnis wahr ist.

Bei einer logischen ODER-Verknüpfung wird die Auswertung abgebrochen, sobald der erste Ausdruck wahr ergibt. Denn dann wird auch zwangsläufig das Gesamtergebnis wahr.

Wenn Sie dagegen die Operatoren `&` und `|` benutzen, werden sämtliche Ausdrücke ausgewertet. Sie sollten daher – wann immer möglich – die Operatoren `&&` und `||` für logische Verknüpfungen verwenden. Damit wird zum einen Ihr Programm etwas schneller, zum anderen droht keine Gefahr, dass Sie die Operatoren für die logischen Verknüpfungen mit den Operatoren für die bitweisen Verknüpfungen verwechseln.



1.3 Kombination von logischen Operatoren mit Vergleichsoperatoren

Die logischen Operatoren und die Vergleichsoperatoren können Sie beliebig miteinander kombinieren. So lassen sich zum Beispiel Ausdrücke wie „A ist kleiner als B und größer als C“ oder „A ist gleich B und B ist ungleich C“ erstellen.

Schauen wir uns dazu einmal das folgende Beispielprogramm an. Es liest drei Zahlen ein und verknüpft dann die Werte dieser Zahlen über Ausdrücke:

```
/* #####
Kombination von logischen Operatoren mit Vergleichsoperatoren
##### */

using System;

namespace Cshp03d_01_05
{
    class Program
    {
        static void Main(string[] args)
        {
            int wertA, wertB, wertC;
            bool ergebnis;

            //Eingabeteil
            Console.Write("Geben Sie den Wert für A ein: ");
            wertA = Convert.ToInt32(Console.ReadLine());

            Console.Write("Geben Sie den Wert für B ein: ");
            wertB = Convert.ToInt32(Console.ReadLine());

            Console.Write("Geben Sie den Wert für C ein: ");
            wertC = Convert.ToInt32(Console.ReadLine());

            //Ausgabeteil
            ergebnis = (wertA < wertB) && (wertB < wertC);
            //bitte in einer Zeile eingeben
            Console.WriteLine("A ist kleiner als B und B ist kleiner
als C: {0}", ergebnis);

            ergebnis = (wertA < wertB) || (wertB < wertC);
            //bitte in einer Zeile eingeben
            Console.WriteLine("A ist kleiner als B oder B ist kleiner
als C: {0}", ergebnis);

            ergebnis = (wertA == wertB) || !(wertB < wertC);
            //bitte in einer Zeile eingeben
            Console.WriteLine("A ist gleich B oder B ist nicht kleiner
als C: {0}", ergebnis);
        }
    }
}
```

Code 1.5: Kombination von logischen Operatoren mit Vergleichsoperatoren

Gehen wir den Code der Reihe nach durch:

Zunächst werden drei Variablen vom Typ `int` vereinbart und eine vom Typ `bool`. Das kennen Sie bereits.

Mit den folgenden Anweisungen lesen wir dann Werte für die drei Variablen `wertA`, `wertB` und `wertC` über die Tastatur ein. Das ist ebenfalls nichts Neues.

Zur Auffrischung:

Die Anweisung `Console.ReadLine()` liefert eine Zeichenkette vom Typ `string` zurück. Diese Zeichenkette müssen Sie ausdrücklich über die Anweisung `Convert.ToInt32()` in eine Zahl konvertieren.

Bitte beachten Sie, dass das Programm abstürzt, wenn Sie für einen der Werte keine ganze Zahl eingeben oder beim Einlesen einfach nur die Eingabetaste drücken.



Interessanter wird es dann im Ausgabeteil. Hier weisen wir der Variablen `ergebnis` dreimal den Wert unterschiedlicher Ausdrücke zu und lassen das Ergebnis mit einem beschreibenden Text auf dem Bildschirm ausgeben.

Der erste Ausdruck ist `(wertA < wertB) && (wertB < wertC)`. Er ist dann wahr, wenn sowohl `wertA < wertB` als auch `wertB < wertC` gilt. Der Compiler wertet zunächst die beiden Vergleiche in den Klammern aus und verknüpft die Ergebnisse dann mit einem logischen UND. Wenn Sie zum Beispiel die Zahlen 10 für `wertA`, 11 für `wertB` und 12 für `wertC` eingeben, erfolgt die Auswertung so:

`10 < 11` – also ist der linke Teil wahr.

`11 < 12` – also ist der rechte Teil auch wahr.

Wahr `&&` wahr ergibt wieder wahr. Damit ist also auch das Ergebnis des gesamten Ausdrucks wahr.

Nehmen wir drei andere Zahlen als Beispiel: `wertA` bekommt den Wert 10, `wertB` den Wert 9 und `wertC` den Wert 12. Die Auswertung sieht dann so aus:

`10 < 9` – der Ausdruck ist falsch.

`9 < 12` – also ist der rechte Teil wahr.

Falsch `&&` wahr ergibt falsch. Also ist das Gesamtergebnis falsch.

Noch einmal zur Erinnerung:

Intern wird der zweite Ausdruck gar nicht mehr überprüft. Da bereits der erste Ausdruck falsch ist, ist auch das Ergebnis der gesamten UND-Verknüpfung falsch. Der Wert der folgenden Ausdrücke spielt dabei keine Rolle.



Der zweite Ausdruck `(wertA < wertB) || (wertB < wertC)` unterscheidet sich vom ersten Ausdruck nur durch den Operator `||`. Hier werden die beiden Ausdrücke also logisch ODER verknüpft. Das Ergebnis dieses Ausdrucks ist dann wahr, wenn entweder der linke Teil oder der rechte Teil wahr ist oder wenn beide Teile wahr sind.

Schauen wir uns auch diesen Ausdruck mit den Zahlen von oben an. Beginnen wir mit den Werten 10 für `wertA`, 11 für `wertB` und 12 für `wertC`.

$10 < 11$ – also ist der linke Teil wahr.

$11 < 12$ – also ist der rechte Teil auch wahr.

Wahr `||` wahr ergibt wieder wahr. Also ist auch hier das Ergebnis wahr, obwohl wir den Operator `||` benutzt haben.



Auch hier wird der zweite Ausdruck intern nicht weiter überprüft. Da bereits der erste Ausdruck wahr ist, spielt das Ergebnis der weiteren Ausdrücke bei einer logischen ODER-Verknüpfung keine Rolle mehr.

Schauen wir uns jetzt die Werte 10 für `wertA`, 9 für `wertB` und 12 für `wertC` an.

$10 < 9$ – der Ausdruck ist falsch.

$9 < 12$ – also ist der rechte Teil wahr.

Falsch `||` wahr ergibt wahr. Also ist auch hier das Ergebnis wahr.

Der dritte Ausdruck `(wertA == wertB) || !(wertB < wertC)` enthält einen NICHT-Operator, der auf den gesamten rechten Teil wirkt. Der Ausdruck `wertB < wertC` wird damit negiert – also umgedreht.

Mit den Werten 10 für `wertA`, 11 für `wertB` und 12 für `wertC` würde die Auswertung so laufen:

$10 == 11$ ist falsch. Also ist der linke Teil falsch.

$11 < 12$ ist wahr und wird negiert. Damit ist der rechte Teil ebenfalls falsch.

Falsch `||` falsch ergibt falsch – also ist das Gesamtergebnis falsch.

Wenn wir die Werte 10 für `wertA`, 9 für `wertB` und 12 für `wertC` verwenden, sieht die Auswertung für den letzten Ausdruck so aus:

$10 == 9$ ist falsch. Damit ist der linke Teil falsch.

$9 < 12$ ist wahr und wird negiert. Damit ist der rechte Teil ebenfalls falsch.

Auch hier ist das Gesamtergebnis wieder falsch.

Damit Sie ein Gefühl für logische Ausdrücke bekommen, sollten Sie jetzt ein wenig mit dem Programm aus dem vorigen Code experimentieren. Geben Sie unterschiedliche Zahlen ein und ersetzen Sie die vorgegebenen Ausdrücke zum Beispiel durch die folgenden Ausdrücke:

```
(wertA != wertB) && (wertA < wertB)
(wertA <= wertB) || (wertA < wertC)
((wertA != wertB) && (wertA < wertB)) || (wertB == wertC)
```

Versuchen Sie nachzuvollziehen, was diese Ausdrücke bedeuten. Der letzte Ausdruck sieht auf den ersten Blick kompliziert aus, sollte Ihnen aber keine großen Schwierigkeiten bereiten, wenn Sie ihn in Umgangssprache „übersetzen“ und die Klammern beachten.

Wenn Sie mit dem Ausdruck nicht klarkommen: Er bedeutet: $((\text{wertA} \neq \text{wertB}) \text{ UND } (\text{wertA} < \text{wertB})) \text{ ODER } (\text{wertB} = \text{wertC})$.

Zusammenfassung

Mit Vergleichsoperatoren können Sie Daten – zum Beispiel Zahlen oder Werte in Variablen – vergleichen.

Mit logischen Operatoren können Sie Aussagen miteinander verknüpfen.

Das Ergebnis eines Vergleichs oder einer logischen Verknüpfung ist entweder wahr oder falsch.

Vergleichsoperatoren und logische Operatoren können Sie nahezu beliebig miteinander kombinieren.

Aufgaben zur Selbstüberprüfung

Überprüfen Sie nun bitte Ihr neu erworbenes Wissen. Lösen Sie die Aufgaben zunächst selbstständig und vergleichen Sie anschließend Ihre Lösungen mit den Angaben im Anhang.

- 1.1 Mit welchem Operator können Sie überprüfen, ob zwei Zahlen gleich sind? Mit welchem Operator können Sie überprüfen, ob eine Zahl größer ist als eine andere?

- 1.2 Geben Sie für die folgenden Ausdrücke jeweils an, ob das Ergebnis wahr oder falsch ist:

a) $1 \leq 2$

b) $1 < 1$

c) `2 != 1`

d) `wahr && falsch`

e) `wahr || falsch`

f) `wahr && !(2 < 1)`

g) `1 < 0 || !(2 != 0)`

- 1.3 Erstellen Sie eine Wahrheitstabelle für den Ausdruck `!(A || B)`. Beispiele für solche Wahrheitstabellen finden Sie im Kapitel.

2 Kontrollstrukturen

*Mit den logischen Ausdrücken aus den letzten Kapiteln konnten wir bisher vor allem die Ergebnisse auf dem Bildschirm ausgeben lassen. Das ist natürlich nicht sonderlich spannend. Richtig interessant werden logische Ausdrücke, wenn Sie sie benutzen, um den Ablauf Ihrer Programme über **Kontrollstrukturen** zu steuern. Dann können Sie nämlich die Ausführung von Anweisungen an Bedingungen knüpfen oder auch Anweisungen gezielt wiederholen. Wie das genau funktioniert, lernen Sie in diesem Kapitel.*

Normalerweise werden Anweisungen bei einfachen Programmen exakt in der Reihenfolge abgearbeitet, in der sie im Quelltext stehen. Über Kontrollstrukturen können Sie die Reihenfolge aber auch verändern – zum Beispiel die Ausführung einer Anweisung von einer Bedingung abhängig machen oder Anweisungen gezielt wiederholen.



2.1 Einfache Verzweigung mit if

Beginnen wir mit der einfachsten Kontrollstruktur – der einfachen `if`-Verzweigung. Eine `if`-Verzweigung (übersetzt etwa „wenn ... dann“) wird eingesetzt, wenn eine oder mehrere Anweisungen abhängig von einer Bedingung ausgeführt werden sollen.

Schauen wir uns die `if`-Verzweigung zunächst an einem Beispiel aus der Umgangssprache an. Dargestellt wird ein Tagesablauf:

Aufstehen

Arbeiten

Nach Hause gehen

WENN die Sonne scheint, **DANN** joggen

Abendessen kochen

Die `if`-Verzweigung finden Sie hier bei der vierten Aktion: **WENN** die Sonne scheint, **DANN** joggen.

Wenn die Sonne nicht scheint – die Bedingung also nicht zutrifft, wird die Aktion „joggen“ nicht ausgeführt, sondern es geht direkt mit der fünften Aktion weiter: Das Abendessen wird gekocht.

Diese einfache `if`-Verzweigung lässt sich grafisch so darstellen:

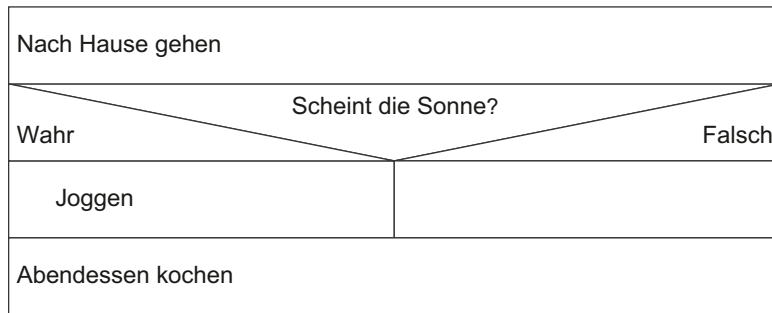


Abb. 2.1: Eine einfache `if`-Verzweigung

Hinweis:

Für die grafische Darstellung der Kontrollstrukturen benutzen wir in diesem Studienheft Nassi-Shneiderman-Diagramme – auch Struktogramme genannt. Andere Formen für die Darstellung von Programmabläufen – wie zum Beispiel Aktivitätsdiagramme der UML – werden Sie später kennenlernen, wenn wir unsere Beispielanwendung entwickeln.

Schauen wir uns jetzt die `if`-Verzweigung in einem C#-Programm an. Der folgende Code soll ermitteln, ob eine eingegebene Zahl größer ist als 5. Damit Sie die `if`-Verzweigung sofort erkennen, haben wir sie im Code durch einen Kommentar markiert und fett hervorgehoben.

```

/* #####
Einfache if-Verzweigung
##### */

using System;

namespace Cshp03d_02_01
{
    class Program
    {
        static void Main(string[] args)
        {
            int zahl;

            Console.Write("Bitte geben Sie eine Zahl ein: ");
            zahl = Convert.ToInt32(Console.ReadLine());
            //die Verzweigung
            if (zahl > 5)
                //bitte in einer Zeile eingeben
                Console.WriteLine("Sie haben eine Zahl größer als 5
                eingegeben.");

            Console.WriteLine("Die Zahl war {0}.", zahl);
        }
    }
}

```

Code 2.1: Einfache `if`-Verzweigung

Die meisten Anweisungen dürften Ihnen bekannt sein. Neu sind nur die Zeilen

```
if (zahl > 5)
    Console.WriteLine("Sie haben eine Zahl größer als 5
    eingegeben.");
```

Schauen wir sie uns etwas genauer an:

In der ersten Zeile folgt nach dem Schlüsselwort `if` in runden Klammern der Ausdruck, der die Verzweigung steuert.

Zur Erinnerung:

Bei den Schlüsselwörtern handelt es sich um reservierte Wörter, die vom Compiler für bestimmte Zwecke verwendet werden – zum Beispiel `using` oder eben `if`. Schlüsselwörter werden in der Standardeinstellung vom Editor in blauer Schrift dargestellt.



In unserem Beispiel überprüfen wir mit dem Ausdruck `(zahl > 5)`, ob der Wert der Variablen `zahl`, den wir mit der Anweisung davor über die Tastatur eingelesen haben, größer ist als 5.

Der Ausdruck in einer `if`-Verzweigung – die **Bedingung** – muss entweder logisch wahr oder logisch falsch zurückliefern. Beispiele für solche Ausdrücke haben Sie im vorigen Kapitel kennengelernt.



Falls die Bedingung wahr ist, wird die nachfolgende Anweisung ausgeführt. In unserem Beispiel wird also der Satz `Sie haben eine Zahl größer als 5 eingegeben` ausgegeben.

Beachten Sie bitte, dass es keine Rolle spielt, ob die Anweisung, die ausgeführt werden soll, in der nächsten oder in derselben Zeile wie die Bedingung steht. Sie könnten die `if`-Verzweigung aus dem vorigen Code also auch so schreiben:

```
if (zahl > 5) Console.WriteLine("Sie haben eine Zahl größer als
5 eingegeben.");
```

Bitte beachten Sie unbedingt:

Sie dürfen **kein** Semikolon hinter den Ausdruck in einer `if`-Anweisung setzen – auch dann nicht, wenn Sie mehr als eine Zeile benutzen. Andernfalls werden nämlich die `if`-Anweisung und die dazugehörige Anweisung getrennt.

Probieren Sie das einfach einmal in dem vorigen Code aus. Setzen Sie hinter die Zeile mit der `if`-Anweisung ein Semikolon und lassen Sie das Programm dann ausführen. Sie werden sehen, die Ausgabe `Sie haben eine Zahl größer als 5 eingegeben` erscheint dann immer – auch wenn Sie eine Zahl kleiner oder gleich 5 eingeben.

Wenn Sie nach der Änderung genau hinsehen, werden Sie auch eine Warnung des Compilers finden, dass möglicherweise eine falsche leere Anweisung vorliegt.



Sie können in der `if`-Verzweigung auch mehr als eine Anweisung ausführen lassen, wenn der Ausdruck wahr ist. Dazu müssen Sie die Anweisungen mit geschweiften Klammern `{ }` in einem **Anweisungsblock** zusammenfassen.

Sehen wir uns dazu das folgende Beispiel an. Es errechnet in einem Anweisungsblock zusätzlich noch die Differenz zwischen dem eingegebenen Wert und dem Wert 5:

```
/* #####
if-Verzweigung mit Anweisungsblock
##### */

using System;

namespace Cshp03d_02_02
{
    class Program
    {
        static void Main(string[] args)
        {
            int zahl;

            Console.Write("Bitte geben Sie eine Zahl ein: ");
            zahl = Convert.ToInt32(Console.ReadLine());

            //die Verzweigung, diesmal mit einem Anweisungsblock
            if (zahl > 5)
            {
                //bitte jeweils in einer Zeile eingeben
                Console.WriteLine("Sie haben eine Zahl größer als 5
eingegeben.");
                Console.WriteLine("Die Differenz zwischen {0} und 5 ist
{1}.", zahl, zahl-5);
            }
            Console.WriteLine("Die Zahl war {0}.", zahl);
        }
    }
}
```

Code 2.2: `if`-Verzweigung mit Anweisungsblock

Wie Sie sehen, handelt es sich beim vorigen Code um eine leicht veränderte Variante des Code 2.1. Es ist eine zweite Anweisung hinzugekommen, die nur innerhalb der `if`-Verzweigung ausgeführt wird. Zusammen mit der bereits im Code 2.1 vorhandenen Anweisung bilden die beiden Anweisungen jetzt einen Anweisungsblock:

```
{
    Console.WriteLine("Sie haben eine Zahl größer als 5
eingegeben.");
    Console.WriteLine("Die Differenz zwischen {0} und 5 ist {1}.",
zahl, zahl-5);
}
```


Die geschweiften Klammern sorgen dabei dafür, dass die beiden Anweisungen vom Compiler als ein logischer Block betrachtet werden. Sie werden nur dann ausgeführt, wenn die Bedingung wahr ist. Wenn Sie die Klammern weglassen, nehmen Sie die zweite Anweisung aus dem Block heraus. Sie wird dann immer ausgeführt – egal, welchen Wert die Bedingung hat.

Achten Sie daher bei Anweisungsblöcken sehr sorgfältig auf die Position der geschweiften Klammern. Denken Sie bitte auch daran, dass für jede öffnende Klammer { auch eine schließende Klammer } vorhanden sein muss.



Fassen wir die Syntax der `if`-Verzweigung noch einmal zusammen.

Tab. 2.1: Syntax der `if`-Verzweigung

<pre>if (Ausdruck) Anweisung;</pre>	<pre>if (Ausdruck) { Anweisung1; Anweisung2; ... }</pre>
---	--

Hinter dem Schlüsselwort `if` steht in runden Klammern der Ausdruck, durch den die Verzweigung gesteuert wird – die Bedingung. Die Bedingung liefert entweder wahr oder falsch. Hinter dem Ausdruck folgt entweder eine Anweisung (links in der Tabelle) oder ein Anweisungsblock (rechts in der Tabelle).

Exkurs: Die Tücken der Operatoren `=` und `==`

Am Code 2.2 können Sie auch noch einmal selbst ausprobieren, wie unangenehm ein Verwechseln der Operatoren `=` und `==` sein kann. Ersetzen Sie den Operator `>` in dem Ausdruck `(zahl > 5)` durch den Operator `==`. Dann werden die Anweisungen in dem Block nur noch ausgeführt, wenn Sie die Zahl 5 eingeben.

Löschen Sie dann eins der Gleichheitszeichen in dem Ausdruck. Sie werden sehen, der Compiler beschwert sich sofort, dass er einen `int`-Typ nicht in einen `bool`-Typ umwandeln kann. Denn der Ausdruck `zahl = 5` liefert ja nicht `true` oder `false` als Ergebnis, sondern den numerischen Wert 5.

Solche Fehler sind gerade mit wenig Programmiererfahrung schnell passiert. Achten Sie daher sorgfältig darauf, dass Sie in Vergleichen mit dem Operator `==` arbeiten, und sehen Sie sich im Zweifelsfall die Fehlermeldung von Visual Studio genau an.

So viel zur einfachen `if`-Verzweigung. Kommen wir jetzt zur `if ... else`-Verzweigung.

2.2 Alternative Verzweigung mit if ... else

Die if ... else-Verzweigung (übersetzt etwa „wenn ... dann – sonst“) wird eingesetzt, wenn Sie neben den Anweisungen, die ausgeführt werden sollen, wenn die Bedingung wahr ist, auch Anweisungen benötigen, die nur dann ausgeführt werden sollen, wenn die Bedingung **falsch** ist.

Erweitern wir das Beispiel mit dem Tagesablauf aus dem letzten Kapitel einmal um eine if ... else-Verzweigung. Es könnte dann so aussehen:

Aufstehen

Arbeiten

Nach Hause gehen

WENN die Sonne scheint, **DANN** joggen, **SONST** fernsehen

Abendessen kochen

Im Unterschied zur einfachen if-Verzweigung wird hier in jedem Fall eine „Freizeitaktion“ ausgeführt. Die Entscheidung wird davon abhängig gemacht, ob die Sonne scheint. Wenn die Bedingung „Sonne scheint“ erfüllt ist, wird die Aktion „joggen“ ausgeführt. Wenn die Bedingung „Sonne scheint“ dagegen nicht erfüllt ist, wird die Aktion „fernsehen“ ausgeführt.

Grafisch lässt sich diese if ... else-Verzweigung so darstellen:

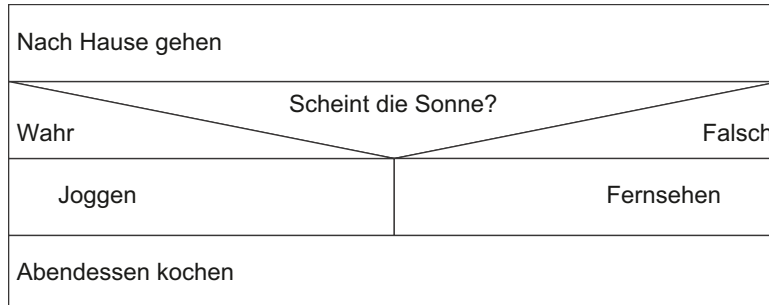


Abb. 2.2: Eine if ... else-Verzweigung

Jetzt fragen Sie sich vielleicht: „Warum soll ich einen eigenen SONST-Zweig einbauen, wenn ich auch eine Anweisung direkt nach dem WENN verwenden kann? Diese Anweisung direkt nach dem WENN wird doch auch ausgeführt, wenn der Ausdruck nicht zutrifft.“

Der Unterschied zwischen den beiden Konstruktionen ist zwar klein, aber bei der Ausführung des Programms erheblich. Sehen wir uns den Tagesablauf einmal an, wenn die Aktion „fernsehen“ nicht im SONST-Zweig steht, sondern als eigene Anweisung direkt hinter dem WENN-Zweig:

Aufstehen

Arbeiten

Nach Hause gehen

WENN die Sonne scheint, DANN joggen

Fernsehen

Abendessen kochen

Wenn Sie diesen Tagesablauf nachspielen, werden Sie den Unterschied sofort sehen: Die Aktion „fernsehen“ ist jetzt völlig unabhängig von der Bedingung bei WENN. Sie wird **immer** ausgeführt – egal, ob die Bedingung zutrifft oder nicht.

Noch deutlicher wird der Unterschied in der grafischen Darstellung. Links sehen Sie den Ablauf mit `if ... else`, rechts dagegen den Ablauf nur mit `if`.

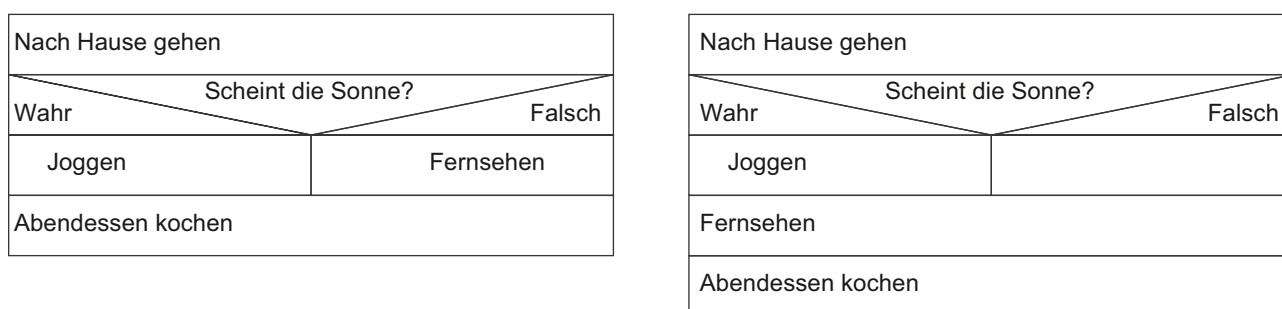


Abb. 2.3: Vergleich von `if ... else` (links) und `if` (rechts)

Achten Sie vor allem darauf, welche Anweisung jeweils nach der Aktion „joggen“ ausgeführt wird. Links wird nach dem Joggen das Abendessen gekocht, rechts dagegen wird nach dem Joggen erst einmal ferngesehen.

Sehen wir uns jetzt die `if ... else`-Verzweigung in einem praktischen Beispiel an. Das folgende Programm erweitert den Code 2.1 und gibt jetzt auch eine Meldung aus, wenn eine Zahl kleiner oder gleich 5 eingegeben wird.

```
/* #####
if ... else-Verzweigung
##### */

using System;

namespace Cshp03d_02_03
{
    class Program
    {
        static void Main(string[] args)
        {
            int zahl;

            Console.Write("Bitte geben Sie eine Zahl ein: ");
            zahl = Convert.ToInt32(Console.ReadLine());

            //die Verzweigung, diesmal mit else
            if (zahl > 5)
                //bitte in einer Zeile eingeben
                Console.WriteLine("Sie haben eine Zahl größer als 5
                eingegeben.");
        }
    }
}
```

```

        else
            //bitte in einer Zeile eingeben
            Console.WriteLine("Sie haben eine Zahl kleiner als 5
            oder 5 eingegeben.");
            Console.WriteLine("Die Zahl war {0}.", zahl);
        }
    }
}

```

Code 2.3: if ... else-Verzweigung

Wie schon bei der einfachen if-Verzweigung können Sie auch hinter else sowohl eine einzelne Anweisung als auch einen Anweisungsblock angeben. Der Anweisungsblock muss dann aber wieder mit den Klammern {} gekennzeichnet werden.

Die Syntax der if ... else-Verzweigung finden Sie noch einmal zusammengefasst in der folgenden Tabelle.

Tab. 2.2: Syntax der if ... else-Verzweigung

if (Ausdruck)	if (Ausdruck)
Anweisung;	{
else	Anweisung1;
Anweisung;	Anweisung2;
	...
	}
	else
	{
	Anweisung1;
	Anweisung2;
	...
	}

2.3 Geschachtelte Verzweigungen

Sie können if-Verzweigungen auch schachteln – also in einer if-Verzweigung mit einer weiteren if-Verzweigung eine weitere Bedingung prüfen. Eine geschachtelte if-Verzweigung könnte so aussehen:

```

if (Ausdruck1)
    if (Ausdruck2)
        Anweisung1;

```

Zuerst wird überprüft, ob Ausdruck1 wahr ist. Wenn das der Fall ist, wird überprüft, ob Ausdruck2 ebenfalls wahr ist. Wenn auch das der Fall ist, wird die Anweisung ausgeführt.

Ist Ausdruck1 dagegen falsch, wird Ausdruck2 gar nicht erst überprüft. Das lässt sich auch sehr einfach in der folgenden Abbildung nachverfolgen.

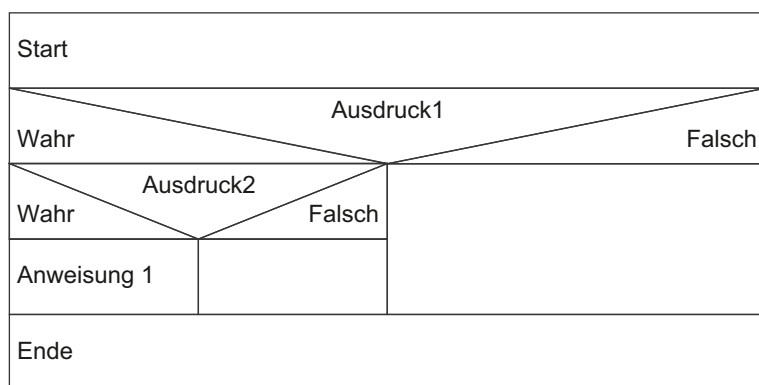


Abb. 2.4: Eine geschachtelte Verzweigung

Da bei einer geschachtelten Verzweigung beide Bedingungen wahr sein müssen, können Sie sie in vielen Fällen auch durch eine logische UND-Verknüpfung ersetzen. Für das Beispiel von oben könnte diese Verknüpfung zum Beispiel so aussehen:

```
if (Ausdruck1 && Ausdruck2)
    Anweisung1;
```

Hier wird ebenfalls überprüft, ob sowohl `Ausdruck1` als auch `Ausdruck2` wahr sind. Nur dann wird die Anweisung ausgeführt.

Die Variante mit der logischen UND-Verknüpfung ist zwar etwas schwieriger zu lesen, dafür aber sehr viel kompakter. Wie Sie bereits an diesem sehr einfachen Beispiel sehen, gibt es beim Programmieren häufig mehrere Möglichkeiten, zum Ziel zu kommen.

Eine wichtige praktische Bedeutung hat die folgende Variante, die noch etwas weiter schachtelt:

```
if (Ausdruck1)
    Anweisung1;
else
    if (Ausdruck2)
        Anweisung2;
    else
        Anweisung3;
```

Hier enthält die erste `if`-Anweisung im `else`-Zweig eine weitere `if`-Anweisung – nämlich `if (Ausdruck2)`.

Der Durchlauf durch diese Konstruktion ist etwas komplizierter als bei den anderen Beispielen: Zuerst wird in der ersten Zeile überprüft, ob `Ausdruck1` wahr ist. Wenn das der Fall ist, wird die Anweisung `Anweisung1` ausgeführt. Danach wird der gesamte Block verlassen, da ja nur noch Anweisungen für den `else`-Zweig folgen, der zu `if (Ausdruck1)` gehört. Es wird also weder `Ausdruck2` geprüft noch werden die Anweisungen `Anweisung2` oder `Anweisung3` ausgeführt.

Ist `Ausdruck1` dagegen falsch, wird im `else`-Zweig mit der Anweisung

```
if (Ausdruck2)
```

weitergemacht.

Ist `Ausdruck2` wahr, wird die Anweisung `Anweisung2` ausgeführt. Ist `Ausdruck2` dagegen falsch, wird der `else`-Zweig ausgeführt, der zu `if (Ausdruck2)` gehört – also die Anweisung `Anweisung3`.

Zum leichteren Verständnis des Ablaufs auch hier wieder die grafische Darstellung:

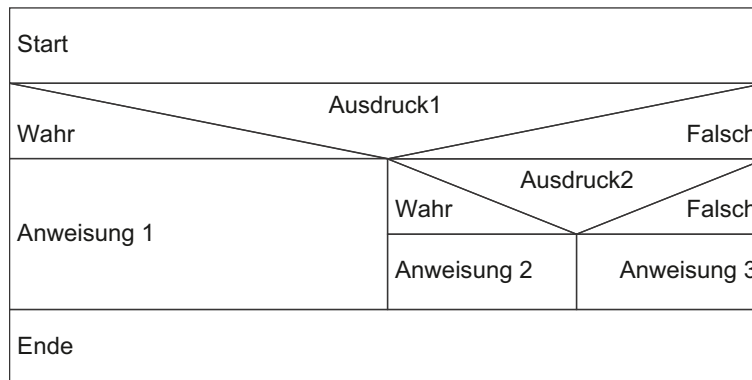


Abb. 2.5: Geschachtelte Verzweigungen mit `if ... else`



Durch die Kombination von geschachtelten `if ... else`-Konstruktionen, in denen die verschiedenen Zweige selbst weitere `if ... else`-Konstruktionen enthalten, lassen sich Entscheidungsketten in einem Programm realisieren.

Sie können die zweite `if`-Anweisung auch direkt hinter das `else` schreiben – zum Beispiel so:

```

if (Ausdruck1)
    Anweisung1;
else if (Ausdruck2)
    ...

```

Achten Sie dann aber darauf, dass Sie die beiden Schlüsselwörter `else` und `if` durch ein Leerzeichen trennen müssen. Ein Schlüsselwort `elseif` kennt C# nicht.

Tipp:


Achten Sie bei allen `if ... else`-Konstruktionen auf die Einrückungen. Damit erhöhen Sie die Lesbarkeit des Quelltextes erheblich. Im folgenden Beispiel sind die Einrückungen zum Beispiel nicht ganz korrekt.

```

if (Ausdruck1)
    Anweisung1;
else
    if (Ausdruck2)
        Anweisung2;
    else
        Anweisung3;

```

Beim flüchtigen Hinsehen entsteht so der Eindruck, als befände sich die zweite if-Anweisung auf der gleichen Ebene wie die erste if-Anweisung. Tatsächlich aber gehört sie zum else-Zweig der ersten if-Anweisung.

Normalerweise setzt der Editor von Visual Studio die Einrückungen automatisch richtig, sobald Sie eine Zeile mit einem if oder einem else abschließen. Sie können die Einrückungen aber auch selbst mit der Taste  setzen.

Sehen wir uns die Entscheidungsketten mit if ... else jetzt an einem Beispiel an. Der folgende Code ermittelt über geschachtelte Abfragen, welche Zahl eingegeben wurde.

```
/* #####
Entscheidungsketten mit if ... else
##### */

using System;

namespace Cshp03d_02_04
{
    class Program
    {
        static void Main(string[] args)
        {
            int zahl;

            Console.Write("Bitte geben Sie eine Zahl ein: ");
            zahl = Convert.ToInt32(Console.ReadLine());

            if (zahl == 5)
                Console.WriteLine("Sie haben die 5 eingegeben.");
            else
                if (zahl < 5)
                    //bitte in einer Zeile eingeben
                    Console.WriteLine("Sie haben eine Zahl kleiner als 5
                    eingegeben.");
                else
                    //bitte in einer Zeile eingeben
                    Console.WriteLine("Sie haben eine Zahl größer als 5
                    eingegeben.");
            Console.WriteLine("Die Zahl war {0}.", zahl);
        }
    }
}
```

Code 2.4: Entscheidungsketten mit if ... else

Schauen wir uns an, wie das Programm genau arbeitet. Zuerst müssen Sie wieder eine Zahl eingeben. Anschließend werden mithilfe der geschachtelten Verzweigung drei Fälle unterschieden.

1. Die Zahl ist gleich 5.
2. Die Zahl ist kleiner als 5.
3. Alles andere – also die Zahl ist größer als 5.

Fall 1 wird durch `if (zahl == 5)` abgedeckt. Sollte `zahl` ungleich 5 sein, geht die erste Verzweigung in ihren `else`-Teil. Dort erfolgt dann die Abfrage für Fall 2 mit `if (zahl < 5)`. Wenn die Zahl auch nicht kleiner als 5 war, geht die zweite Verzweigung in ihren `else`-Teil. Damit wird dann auch Fall 3 abgedeckt.

Zum leichteren Nachvollziehen finden Sie den Ablauf auch noch einmal in der folgenden Abbildung.

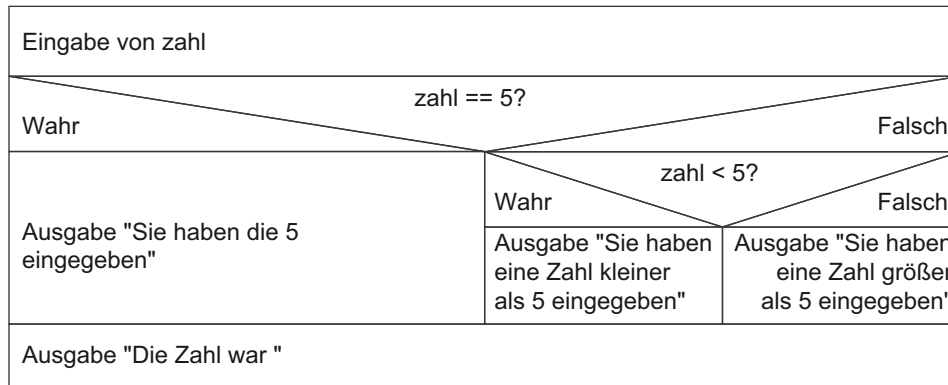


Abb. 2.6: Die Entscheidungskette aus Code 2.4

2.4 Mehrfachauswahl mit `switch ...case`

Wie Sie gerade selbst gesehen haben, sind geschachtelte Verzweigungen recht vielseitig. Allerdings geht schnell der Überblick verloren, wenn Sie nicht sehr sorgfältig arbeiten.

C# kennt aber noch eine Möglichkeit, gezielt auf verschiedene Werte zu reagieren – die Mehrfachauswahl über `switch ... case`. Sie ist wie folgt aufgebaut:

```
switch (Ausdruck)
{
    case Konstante1:
        Anweisung1;
        break;
    case Konstante2:
        Anweisung2;
        break;
    ...
    default:
        Anweisung;
        break;
}
```


Grafisch lässt sich die `switch ... case`-Konstruktion zum Beispiel so darstellen:

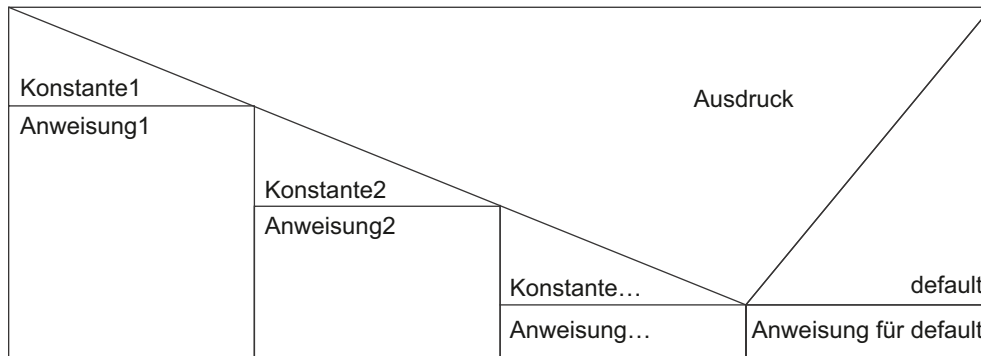


Abb. 2.7: Die `switch ... case`-Konstruktion

Eingeleitet wird die Konstruktion von dem Schlüsselwort `switch`¹. Auf das Schlüsselwort folgt in Klammern entweder ein Wert vom Typ `char`, ein ganzzahliger Typ oder eine Zeichenkette. Einen Typ mit Kommastellen dagegen können Sie nicht verwenden.

Anschließend folgt eine öffnende geschweifte Klammer und es beginnt der **Auswertungsblock**. Im Auswertungsblock stehen die einzelnen Fälle, zwischen denen Sie unterscheiden wollen – die **case-Marken** oder **case-Zweige**.

Jeder Zweig wird dabei einzeln mit dem Schlüsselwort `case`² eingeleitet. Dann folgen eine Konstante und ein Doppelpunkt. Danach kommen die Anweisungen, die in diesem Fall ausgeführt werden sollen. Abgeschlossen werden die Anweisungen für den Zweig durch das Schlüsselwort `break`³ und ein Semikolon.

Ausgeführt wird ein `case`-Zweig, wenn der Ausdruck hinter `switch` mit der Konstante des Zweigs übereinstimmt. Die Ausführung der Anweisungen wird dabei beim nächsten `break` beendet.

Bitte beachten Sie:

Die Anweisungen in den einzelnen `case`-Zweigen müssen nicht von geschweiften Klammern umgeben sein.

Für jede Konstante darf es nur genau einen `case`-Zweig geben. Sie können eine Konstante also nicht mehrfach aufführen.



Bleibt noch das Schlüsselwort `default`⁴ am Ende des Auswertungsblocks. Hier können Sie Anweisungen angeben, die ausgeführt werden, wenn keiner der Fälle davor zutrifft. Der Einsatz von `default` ist optional; das heißt, Sie können `default` auch weglassen. Damit sparen Sie zwar etwas Tipparbeit, laufen aber besonders bei umfangreichen `switch ... case`-Konstruktionen Gefahr, dass das Programm in einen undefinierten Zustand gerät, wenn keiner der `case`-Zweige zutrifft.

1. *Switch* bedeutet übersetzt so viel wie „Schalter“.
2. *Case* bedeutet übersetzt so viel wie „Fall“.
3. *Break* bedeutet übersetzt so viel wie „Unterbrechung“.
4. *Default* bedeutet frei übersetzt so viel wie „Standard“.

Daher sollten Sie den `default`-Zweig vor allem bei Ihren ersten Versuchen immer einbauen – auch wenn Sie hier nur eine Kontrollmeldung ausgeben lassen, dass keiner der `case`-Zweige durchlaufen wurde.

Ein wichtiger Tipp:

Denken Sie an die `break`-Anweisungen am Ende der Zweige. Wenn das `break` fehlt, meldet der Compiler einen Fehler. Achten Sie daher auch darauf, dass Sie nicht nur die `case`-Zweige mit `break` abschließen, sondern auch den `default`-Zweig.

Schauen wir uns die `switch ... case`-Konstruktion jetzt an einem praktischen Beispiel an. Der folgende Code erzeugt eine „Menükarte“ mit drei Einträgen und liefert abhängig von Ihrer Auswahl einen Text zurück.

```
/* #####
switch ... case
##### */

using System;

namespace Cshp03d_02_05
{
    class Program
    {
        static void Main(string[] args)
        {
            char essenWahl;

            Console.WriteLine("Sie haben folgende Auswahl: \n");
            Console.WriteLine("a Schweineschnitzel mit Nudeln");
            Console.WriteLine("b Wiener Schnitzel mit Pommes");
            //bitte in einer Zeile eingeben
            Console.WriteLine("c Vegetarische Hackbällchen mit
            Reis\n");
            Console.Write("Was möchten Sie essen? ");

            essenWahl = Convert.ToChar(Console.Read());

            //die Auswertung von essenWahl
            switch(essenWahl)
            {
                //der case-Zweig für a
                case 'a':
                    //bitte in einer Zeile eingeben
                    Console.WriteLine("Sie haben Schweineschnitzel mit
                    Nudeln gewählt!");
                    Console.WriteLine("Kommt sofort!");
                    break;
                //der case-Zweig für b
                case 'b':
                    //bitte in einer Zeile eingeben
                    Console.WriteLine("Sie haben Wiener Schnitzel mit
                    Pommes gewählt!");
                    Console.WriteLine("Das dauert einen Moment!");
                    break;
```

```

        //der case-Zweig für c
        case 'c':
            //bitte in einer Zeile eingeben
            Console.WriteLine("Sie haben vegetarische Hackbällchen
            mit Reis gewählt!");
            Console.WriteLine("Einen Augenblick!");
            break;
        //für alles andere
        default:
            //bitte in einer Zeile eingeben
            Console.WriteLine("Sie haben keine gültige Auswahl
            getroffen!");
            Console.WriteLine("Dann gibt es eben nichts.");
            break;
    }
}
}
}

```

Code 2.5: switch ... case-Konstruktion

Die drei Gerichte werden im Code durch die drei `case`-Zweige unterschieden. Wenn Sie etwas anderes als a, b oder c eingeben, werden die Anweisungen hinter `default` ausgeführt.

Allerdings werden die Anweisungen im `default`-Block auch dann ausgeführt, wenn Sie zum Beispiel A eingeben. Denn aus Sicht des Programms handelt es sich bei a und A um zwei verschiedene Werte – und lediglich für den Wert a ist ja ein `case`-Zweig vorhanden. Sie könnten nun natürlich auch für die entsprechenden Großbuchstaben getrennte `case`-Zweige erstellen und die Anweisungen dann dort noch einmal aufführen. Es geht aber auch sehr viel komfortabler – nämlich mit einem kleinen „Kniff“.

Ergänzen Sie einfach unmittelbar vor dem `case`-Zweig für den Kleinbuchstaben einen `case`-Zweig für den dazugehörigen Großbuchstaben. Diesen Zweig lassen Sie komplett leer – also ohne Anweisungen und auch ohne `break`. Für die Buchstaben A und a könnte das so aussehen:

```

switch (essenWahl)
{
    case 'A':
    case 'a':
        Console.WriteLine("Sie haben Schweineschnitzel mit Nudeln
        gewählt!");
        Console.WriteLine("Kommt sofort!");
        break;
    case 'b':
        ...
}

```

Wenn der Buchstabe A eingegeben wird, beginnt das Programm mit der Bearbeitung bei der entsprechenden case-Marke – also case 'A':. Dann werden alle Anweisungen bis zum nächsten break ausgeführt – also die Anweisungen, die zur case-Marke case 'a': gehören.



Dieser „Kniff“ funktioniert nur dann, wenn die case-Marken direkt aufeinanderfolgen.

Auch switch ... case-Konstruktionen lassen sich schachteln. Das ist zum Beispiel dann interessant, wenn Sie einen Hauptfall in weitere Unterfälle untergliedern wollen. Im folgenden Code werden zum Beispiel für zwei Gerichte noch Beilagen angeboten. Außerdem haben wir den Code so erweitert, dass jetzt auch Großbuchstaben berücksichtigt werden.

```
/* #####
geschachtelte switch ... case-Konstruktion
##### */

using System;

namespace Cshp03d_02_06
{
    class Program
    {
        static void Main(string[] args)
        {
            char essenWahl, beilagenWahl;

            Console.WriteLine("Sie haben folgende Auswahl: \n");
            Console.WriteLine("a Schweineschnitzel");
            Console.WriteLine("b Wiener Schnitzel\n");
            Console.Write("Was möchten Sie essen? ");

            essenWahl = Convert.ToChar(Console.Read());
            //den Tastaturpuffer leeren
            Console.ReadLine();

            //bitte in einer Zeile eingeben
            Console.WriteLine("Sie können folgende Beilagen wählen: \n");
            Console.WriteLine("c Pommes");
            Console.WriteLine("d Reis\n");
            Console.Write("Welche Beilage möchten Sie? ");

            beilagenWahl = Convert.ToChar(Console.Read());

            //die Auswertung von essenWahl
            switch (essenWahl)
            {
                case 'A':
                case 'a':
```

```

//die Auswertung von beilagenWahl
switch (beilagenWahl)
{
    case 'C':
    case 'c':
        //bitte in einer Zeile eingeben
        Console.WriteLine("Sie haben Schweineschnitzel mit
        Pommes gewählt!");
        break;
    case 'D':
    case 'd':
        //bitte in einer Zeile eingeben
        Console.WriteLine("Sie haben Schweineschnitzel mit
        Reis gewählt!");
        break;
    default:
        Console.WriteLine("Diese Beilage gibt es nicht.");
        break;
} //hier endet die Auswertung von beilagenWahl
break; //hier enden die case-Marken für a und A
case 'B':
case 'b':
    //die Auswertung von beilagenWahl
    switch (beilagenWahl)
    {
        case 'C':
        case 'c':
            //bitte in einer Zeile eingeben
            Console.WriteLine("Sie haben Wiener Schnitzel mit
            Pommes gewählt!");
            break;
        case 'D':
        case 'd':
            //bitte in einer Zeile eingeben
            Console.WriteLine("Sie haben Wiener Schnitzel mit
            Reis gewählt!");
            break;
        default:
            Console.WriteLine("Diese Beilage gibt es nicht.");
            break;
    } //hier endet die Auswertung von beilagenWahl
    break; //hier enden die case-Marken für b und B
default: //für die Auswertung von essenWahl
    //bitte in einer Zeile eingeben
    Console.WriteLine("Dieses Gericht steht nicht auf der
    Karte.");
    break;
} //hier endet die Auswertung von essenWahl
}
}
}

```

Code 2.6: Verschachteltes switch ... case

Damit Sie die verschiedenen Ebenen leichter nachvollziehen können, haben wir den Anfang und das Ende jeweils mit Kommentaren gekennzeichnet. Welche verschiedenen Fälle das Programm abdeckt, zeigt Ihnen die folgende Abbildung:

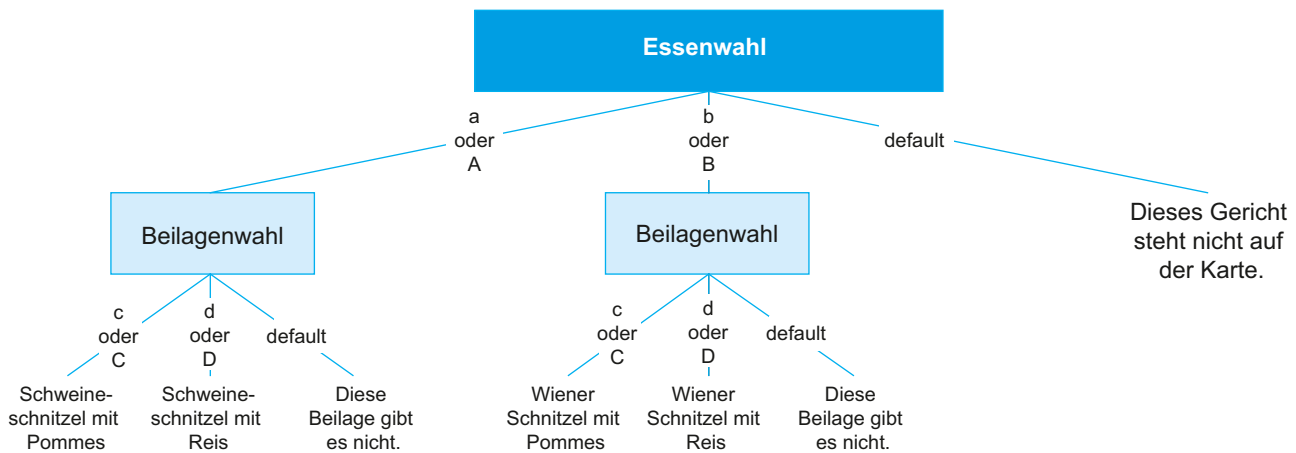


Abb. 2.8: Darstellung der möglichen Fälle aus Code 2.6

Genau wie verschachtelte `if ... else`-Konstruktionen können auch geschachtelte `switch ... case`-Konstruktionen sehr schnell unübersichtlich werden. Versuchen Sie daher, die Schachtelungen auf so wenig Ebenen wie eben möglich zu beschränken.

Zum Schluss dieses Kapitels fassen wir noch einmal ein paar wichtige Hinweise zur `if`-Konstruktion und zur `switch ... case`-Konstruktion zusammen:

- Wenn Sie eine Mehrfachauswahl programmieren wollen, können Sie die `switch ... case`-Konstruktion nur dann verwenden, wenn Sie mit Werten arbeiten, die sich durch eine ganze Zahl, ein Zeichen oder eine Zeichenkette darstellen lassen. Bei Zahlen mit Kommastellen müssen Sie geschachtelte `if ... else`-Verzweigungen benutzen.
- Denken Sie in einer `switch ... case`-Konstruktion an das `break` in den `case`-Zweigen und dem `default`-Zweig!
- Achten Sie peinlich genau darauf, dass die Konstanten in den `case`-Zweigen zum Typ des Ausdrucks bei `switch` passen. Wenn Sie zum Beispiel ganze Zahlen verarbeiten, dürfen Sie diese Zahlen **nicht** mit Apostrophen umfassen. Der Compiler behandelt dann nämlich die Konstanten der `case`-Marken als Zeichen und es gibt keine passende `case`-Marke – egal, was Sie eingeben.

Das folgende Codefragment führt zum Beispiel immer die Anweisungen im `default`-Zweig aus – auch dann, wenn Sie 1 oder 2 eingeben:

```

int zahlWahl;
Console.WriteLine("Geben Sie eine Zahl ein: ");
zahlWahl = Convert.ToInt32(Console.ReadLine());
switch(zahlWahl)
{
    case '1':
        Console.WriteLine("Sie haben 1 eingegeben!");
        break;

```

```
    case '2':
        Console.WriteLine("Sie haben 2 eingegeben!");
        break;
    default:
        Console.WriteLine("Sie haben weder 1 noch 2 eingegeben!");
        break;
}
```

- Achten Sie sehr genau auf die Klammern bei geschachtelten `if ... else`-Konstruktionen. Verwenden Sie lieber zu viele Klammern als zu wenig. Besonders bei starken Schachtelungen besteht gerade zu Beginn die Gefahr, dass Sie den Überblick verlieren. Klammern Sie daher am Anfang am besten jeden Zweig – auch wenn es eigentlich nicht zwingend erforderlich ist.

Denken Sie aber bitte daran: Der Compiler überprüft lediglich die Syntax Ihrer Programme. Das heißt, er zählt mit, ob Sie für jede öffnende Klammer auch eine schließende verwenden. Ob Sie die Klammern logisch an die richtige Stelle setzen, kann der Compiler nicht prüfen.

- Achten Sie bei `if ... else`-Verzweigungen genau darauf, zu welchem `if` ein `else`-Teil gehört.

Zusammenfassung

Mit einer `if`-Verzweigung können Sie Anweisungen abhängig von einer Bedingung ausführen lassen.

Über den `else`-Zweig können Sie bei `if` Anweisungen angeben, die ausgeführt werden sollen, wenn die Bedingung **nicht** zutrifft.

Sowohl im `if`-Zweig als auch im `else`-Zweig können Sie mehrere Anweisungen mit den Klammern `{ }` zu einem Anweisungsblock zusammenfassen.

Für Mehrfachauswahlen bei ganzzahligen Werten, Zeichen oder Zeichenketten können Sie die `switch ... case`-Konstruktion verwenden.

Sowohl die `if`-Anweisung als auch die `switch ... case`-Konstruktion können Sie schachteln.

Aufgaben zur Selbstüberprüfung

- 2.1 Schreiben Sie ein Programm, das zwei ganze Zahlen x und y über die Tastatur einliest und anschließend ausgibt, ob x größer ist als y oder umgekehrt. Benutzen Sie dafür eine `if ... else`-Verzweigung.

- 2.2 Ergänzen Sie die `switch ... case`-Konstruktion in dem folgenden Quelltext. Geben Sie bitte in den Lücken die jeweils erforderliche Anweisung an. Ein `default`-Zweig ist in dem Programm nicht vorgesehen.

```
/* #####
switch ... case mit Lücken
#####*/

using System;

namespace LoesII2
{
    class Program
    {
        static void Main(string[] args)
        {
            int essenWahl;

            Console.WriteLine("Wählen Sie bitte ein Essen aus:
            \n");
            Console.WriteLine("1 Jägerschnitzel mit Pommes");
            Console.WriteLine("2 Currywurst mit Pommes");
            Console.WriteLine("3 Bratwurst mit Brötchen\n");
            Console.WriteLine("Was möchten Sie essen? ");

            essenWahl = Convert.ToInt32(Console.ReadLine());

            switch _____
            {
                _____
            }
        }
    }
}
```



```
        Console.WriteLine("Sie haben ein Jägerschnitzel  
        mit Pommes gewählt.");  
        _____;  
    _____  
    Console.WriteLine("Sie haben eine Currywurst mit  
    Pommes gewählt.");  
    _____  
    case _____:  
        Console.WriteLine("Sie haben eine Bratwurst mit  
        Brötchen gewählt.");  
        _____  
    }  
}  
}  
}
```

3 Schleifen

*Im letzten Kapitel haben Sie gelernt, wie Sie Verzweigungen und Mehrfachauswahlen in Ihren Programmen benutzen. In diesem Kapitel werden Sie ein weiteres wichtiges Konstrukt zum Erstellen anspruchsvoller Programme kennenlernen: die **Schleifen**.*



Mit Schleifen können Sie Teile eines Programms wiederholen lassen. Dabei führt der Rechner die Anweisungen im **Schleifenkörper** wiederholt in Abhängigkeit von **Bedingungen** aus. Die Bedingungen stehen entweder am Anfang der Schleife im **Schleifenkopf** oder am Ende der Schleife im **Schleifenfuß**.

Hinweis:

Die Bedingungen, die die Ausführung einer Schleife steuern, werden auch **Abbruchbedingungen** genannt. C# wertet Abbruchbedingungen allerdings genau andersherum aus, als es dem „normalen“ Sprachgebrauch entspricht. Wenn die Abbruchbedingung wahr ist, wird die Schleife ausgeführt. Ist die Abbruchbedingung dagegen falsch, wird die Schleife nicht ausgeführt. Wir benutzen daher in diesem Kapitel einfach den Ausdruck „Bedingung“.

Grundsätzlich unterscheidet C# drei Schleifentypen:

- die **kopfgesteuerte Schleife**, bei der die Bedingung am Anfang steht,
- die **fußgesteuerte Schleife**, bei der die Bedingung am Ende steht, und
- die **Zählschleife**, einen Sonderfall der kopfgesteuerten Schleife.

Wir werden im Verlauf des Kapitels nacheinander auf diese Schleifentypen eingehen.

Außerdem werden Sie zwei Befehle zur Schleifensteuerung kennenlernen: `break` und `continue`.

Bevor Sie sich mit den Schleifen beschäftigen, noch ein wichtiger Hinweis:

Übernehmen Sie die Programme dieses Kapitels sehr sorgfältig. Bei Fehlern in einer Schleife droht eine **Endlosschleife**. Das heißt, das Programm wiederholt die Schleifenanweisungen immer wieder und kann nicht korrekt beendet werden. Sie müssen es dann „gewaltsam“ abbrechen – zum Beispiel mit der Tastenkombination **Strg** + **C**. Im schlimmsten Fall bleibt Ihnen bei einer Endlosschleife nichts anderes übrig, als den Rechner neu zu starten. Achten Sie deshalb vor allem bei den Bedingungen sorgfältig darauf, dass Ihnen keine Fehler unterlaufen.

Schauen wir uns jetzt die verschiedenen Schleifen in C# der Reihe nach an. Beginnen wir mit der kopfgesteuerten Schleife mit `while`.

3.1 Kopfgesteuerte Schleife mit while

Die kopfgesteuerte Schleife mit `while` ist die einfachste und allgemeinste Schleife von C#.

Sie hat die folgende Syntax:

```
while (Ausdruck) //Schleifenkopf
    Anweisung;    //Schleifenkörper
```

Zuerst wird der Ausdruck im Schleifenkopf ausgewertet. Wenn er wahr ist, wird die Anweisung im Schleifenkörper ausgeführt. Danach springt das Programm wieder zum Schleifenkopf und der Ausdruck wird erneut ausgewertet. Dieser Prozess wiederholt sich so lange, bis der Ausdruck falsch wird. Dann wird das Programm mit der nächsten Anweisung hinter dem Schleifenkörper fortgesetzt.

Bitte beachten Sie:

Hinter dem Schleifenkopf dürfen Sie **kein** Semikolon setzen. Damit trennen Sie den Schleifenkopf von der eigentlichen Schleife und es wird nur noch der Schleifenkopf ausgeführt. Da sich dabei aber der Ausdruck nicht verändern kann, wird der Schleifenkopf endlos wiederholt.



Grafisch lässt sich die `while`-Schleife so darstellen:

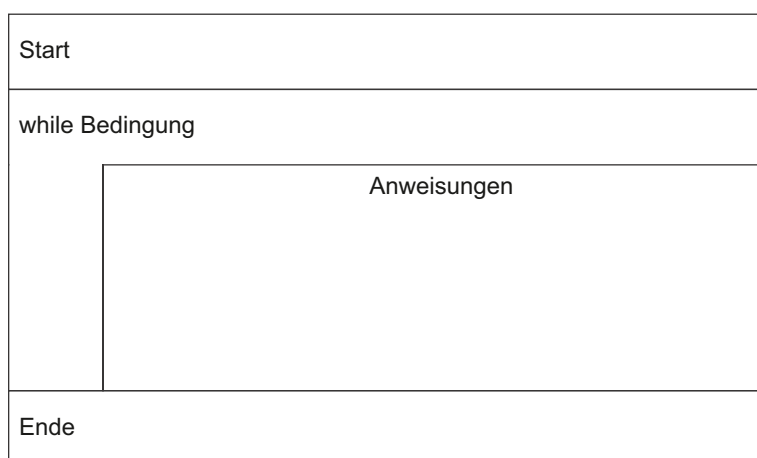


Abb. 3.1: Die `while`-Schleife

Merken Sie sich für die while-Schleife:

Die Schleife wird so lange ausgeführt, wie der Ausdruck im Schleifenkopf wahr ist.



Beachten Sie bitte, dass sich bei der `while`-Schleife die Reihenfolge bei der Ausführung der Anweisungen verändert. Bisher wurden in unseren Programmen immer alle Anweisungen von „oben“ nach „unten“ abgearbeitet. Jetzt tritt zum ersten Mal der Fall ein, dass nach einer Anweisung mit einer Anweisung weitergearbeitet wird, die sich im Quelltext vor der ausgeführten Anweisung befindet – eben mit der Überprüfung der Bedingung für die Schleife.

Wie schon bei der `if`-Verzweigung können Sie auch bei der `while`-Schleife mehrere Anweisungen in einem Anweisungsblock zusammenfassen. Dann sieht die `while`-Schleife so aus:

```
while (Ausdruck) //Schleifenkopf
{
    //Anfang des Schleifenkörpers
    Anweisung1;
    Anweisung2;
    ...;
} //Ende des Schleifenkörpers
```

Sehen wir uns die `while`-Schleife jetzt in einem Programm an. Der folgende Code zählt auf dem Bildschirm bis 10.

```
/* #####
while-Schleife
##### */

using System;

namespace Cshp03d_03_01
{
    class Program
    {
        static void Main(string[] args)
        {
            int schleifenVariable = 0;

            //die Schleife
            while (schleifenVariable <= 10)
            {
                //der aktuelle Wert wird ausgegeben
                //bitte in einer Zeile eingeben
                Console.WriteLine("Die Variable hat jetzt den Wert {0}",
                    schleifenVariable);
                //schleifenVariable wird erhöht
                schleifenVariable++;
            }
        }
    }
}
```

Code 3.1: `while`-Schleife

Schauen wir uns die Schleife im Detail an:

Im Schleifenkopf `while (schleifenVariable <= 10)` wird der Ausdruck `schleifenVariable <= 10` ausgewertet. Solange dieser Ausdruck wahr ist, werden die Anweisungen im Schleifenkörper ausgeführt. In unserem Beispiel läuft die Schleife also, bis `schleifenVariable` den Wert 11 erreicht hat.



Eine Variable, die im Ausdruck einer Schleife überprüft wird, wird auch **Schleifenvariable** genannt.

Im Schleifenkörper selbst werden zwei Anweisungen ausgeführt:

```
Console.WriteLine("Die Variable hat jetzt den Wert {0}",
    schleifenVariable);
schleifenVariable++;
```

Die erste Anweisung gibt den aktuellen Wert von `schleifenVariable` aus. In der zweiten Anweisung wird der Wert von `schleifenVariable` bei jedem Durchlauf um den Wert 1 erhöht.

Zwei Sachen sind bei einer `while`-Schleife sehr wichtig:

- 1) Sie müssen sicherstellen, dass die Schleifenvariable korrekt initialisiert ist – also den richtigen Wert zugewiesen bekommt. Wenn Sie zum Beispiel `schleifenVariable` im vorigen Code mit dem Wert 1 initialisieren, erfolgt ein Schleifendurchlauf weniger.
- 2) Sie müssen den Wert der Schleifenvariablen innerhalb der Schleife verändern. Das übernimmt in unserem Beispiel die Anweisung

```
schleifenVariable++;
```

im Schleifenkörper.

Wenn Sie diese Veränderung nicht vornehmen, ergibt die Auswertung der Bedingung im Schleifenkopf immer wieder denselben Wert – und damit hätten Sie eine Endlosschleife konstruiert.

Probieren Sie das ruhig einmal aus. Kommentieren Sie die Anweisung zum Verändern von `schleifenVariable` aus und lassen Sie das Programm noch einmal ausführen. Denken Sie vor dem Start bitte daran, alle noch nicht gespeicherten Daten zu sichern. Sie werden sehen, das Programm läuft so lange weiter, bis Sie es mit der Tastenkombination **Strg** + **C** abbrechen.

Testen Sie auch einmal, was geschieht, wenn Sie hinter den Schleifenkopf im vorigen Code ein Semikolon setzen. Nach dem Start des Programms erscheint dann nur ein leeres Fenster für die Eingabeaufforderung mit einer blinkenden Einfügemarke. Scheinbar passiert also gar nichts. Tatsächlich aber läuft die Schleife endlos – allerdings wird lediglich der Ausdruck immer wieder überprüft. Wenn Sie genau hinsehen, erkennen Sie auch, dass der Compiler Ihnen eine entsprechende Warnung anzeigt – nämlich zu einer falschen leeren Anweisung.

So viel zur kopfgesteuerten Schleife mit `while`.

3.2 Fußgesteuerte Schleife mit `do ... while`

Die `do ... while`-Schleife arbeitet ähnlich wie die `while`-Schleife. Allerdings erfolgt die Auswertung der Bedingung erst im Schleifenfuß – daher auch der Name **fußgesteuerte Schleife**.

Die Syntax der `do ... while`-Schleife sieht so aus:

```
do                                //Schleifenkopf
    Anweisung;                   //Schleifenkörper

while (Ausdruck);                //Schleifenfuß
```

Zuerst wird die Anweisung im Schleifenkörper ausgeführt. Anschließend wird der Ausdruck im Schleifenfuß geprüft. Wenn dieser Ausdruck wahr ist, wird die Anweisung im Schleifenkörper wiederholt, andernfalls wird die Anweisung nach dem Schleifenfuß ausgeführt.



Bitte beachten Sie:

Bei einer `do ... while`-Schleife **müssen** Sie den Schleifenfuß mit einem Semikolon abschließen. Hier ist ja die gesamte Schleife zu Ende. Wenn Sie das Semikolon weglassen, meldet der Compiler einen Syntaxfehler.

Da die Überprüfung der Bedingung erst am Ende der Schleife erfolgt, wird der Schleifenkörper bei der `do ... while`-Schleife – anders als bei der `while`-Schleife – mindestens einmal durchlaufen. Eine fußgesteuerte Schleife sollten Sie daher immer dann einsetzen, wenn Sie sichergehen wollen, dass die Schleife mindestens einmal abgearbeitet wird.



Merken Sie sich für die `do ... while`-Schleife:

Die Schleife wird in jedem Fall mindestens einmal durchlaufen.

Grafisch lässt sich die `do ... while`-Schleife so darstellen:

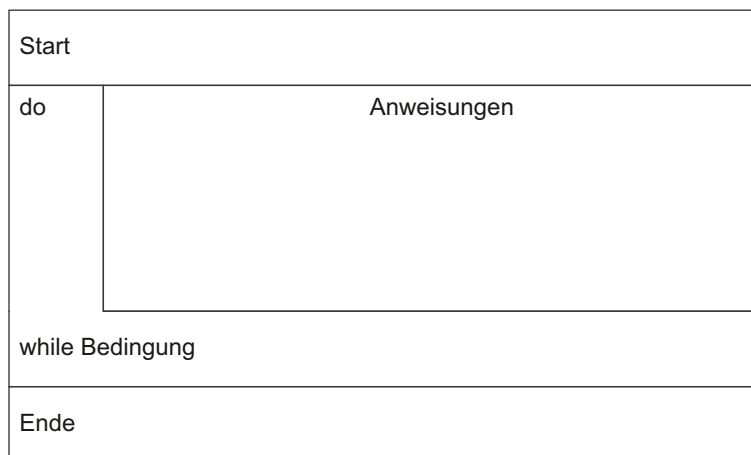


Abb. 3.2: Die `do ... while`-Schleife

Auch im Schleifenkörper der `do ... while`-Schleife können Sie einen Anweisungsblock verwenden. Dann sieht die Syntax so aus:

```
do                                //Schleifenkopf
{                                //Anfang des Schleifenkörpers
    Anweisung1;
    Anweisung2;
    ...;
} while(Ausdruck); //Ende des Schleifenkörpers und Schleifenfuß
```

Schauen wir uns auch die `do ... while`-Schleife an einem Beispiel an. Im folgenden Code werden so lange Zahlen eingelesen, bis Sie einen Wert größer als 10 eingeben.

```
/* #####
do ... while-Schleife
##### */

using System;

namespace Cshp03d_03_02
{
    class Program
    {
        static void Main(string[] args)
        {
            int schleifenVariable = 0;

            do
            {
                //bitte in einer Zeile eingeben
                Console.WriteLine("Bitte geben Sie einen Wert größer als 10
                ein. ");
                schleifenVariable = Convert.ToInt32(Console.ReadLine());
            } while (schleifenVariable <= 10);
            Console.WriteLine("Danke.");
        }
    }
}
```

Code 3.2: do ... while-Schleife

Im Schleifenkörper finden Sie eine Aufforderung zur Eingabe einer Zahl und die Anweisung zum Einlesen der Zahl. Diese Anweisungen werden so lange ausgeführt, bis der Ausdruck `schleifenVariable <= 10` falsch wird – also bis eine Zahl größer als 10 eingegeben wird.

Die Veränderung der Variablen `schleifenVariable` erfolgt in unserem Beispiel durch das Einlesen des Wertes innerhalb der Schleife. Damit ist also sichergestellt, dass keine Endlosschleife entstehen kann – außer der Anwender gibt immer wieder einen Wert kleiner als 11 ein.

Da die Schleife im vorigen Code mindestens einmal durchlaufen wird, könnten Sie hier auch auf die Initialisierung der Schleifenvariablen vor der Schleife verzichten. Die Initialisierung würde ja beim ersten Durchlauf der Schleife erfolgen. Damit wäre auch sichergestellt, dass die Schleifenvariable bei der ersten Auswertung des Ausdrucks im Schleifenfuß einen definierten Wert hat.

Grundsätzlich lässt sich jede kopfgesteuerte Schleife in eine fußgesteuerte ändern und umgekehrt. Beim vorigen Code ist diese Änderung ganz einfach. Sie können die `while`-Anweisung, die im Schleifenfuß steht, in den Schleifenkopf schreiben und lassen das `do` weg. Denken Sie dabei bitte aber auch daran, das Semikolon hinter dem Ausdruck bei `while` zu löschen.

Weitere Änderungen sind nicht nötig, weil die Variable `schleifenVariable` zu Beginn mit 0 initialisiert wird und der Ausdruck `schleifenVariable <= 10` damit wahr ist. Der Schleifenkörper wird also in jedem Fall mindestens einmal durchlaufen. Der vorige Code sieht mit einer kopfgesteuerten Schleife so aus:

```
/* #####
Die umgebaute Schleife
##### */

using System;

namespace Cshp03d_03_03
{
    class Program
    {
        static void Main(string[] args)
        {
            int schleifenVariable = 0;

            //der Fuß wird jetzt zum Kopf
            //Denken Sie daran, das Semikolon zu löschen!
            while (schleifenVariable <= 10)
            {
                //bitte in einer Zeile eingeben
                Console.WriteLine("Bitte geben Sie einen Wert größer als 10
                ein. ");
                schleifenVariable = Convert.ToInt32(Console.ReadLine());
            }
            Console.WriteLine("Danke.");
        }
    }
}
```

Code 3.3: Die umgebaute Schleife

Auch die `while`-Schleife aus dem Code 3.1 lässt sich durch eine `do ... while`-Schleife ersetzen.

Bevor Sie weiterlesen ...

Versuchen Sie erst einmal selbst, den Umbau durchzuführen.

Der Code sieht dann so aus:

```
/* #####
Noch eine umgebaute Schleife
##### */

using System;

namespace Cshp03d_03_04
{
    class Program
    {
```



```
static void Main(string[] args)
{
    int schleifenVariable = 0;

    do
    {
        //bitte in einer Zeile eingeben
        Console.WriteLine("Die Variable hat jetzt den Wert {0}",
            schleifenVariable);
        schleifenVariable++;
    } while (schleifenVariable <= 10);
}
}
```

Code 3.4: Noch eine umgebaute Schleife

Ob Sie nun eine `while`- oder `do ... while`-Schleife verwenden, ist in vielen Fällen auch eine Frage des persönlichen Geschmacks. `do ... while`-Schleifen werden aber zum Beispiel häufig für das Einlesen von Werten verwendet, da hier ja die Eingabe des Anwenders bereits vor der Überprüfung der Bedingung zur Verfügung steht.

So viel an dieser Stelle zur `do ... while`-Schleife. Am Ende dieses Kapitels werden wir diese Schleifenform noch einmal in einem Exkurs verwenden und Ihnen zeigen, wie Sie Eingabefehler abfangen können.

3.3 Zählschleife mit `for`

Mit der kopfgesteuerten `while`- und der fußgesteuerten `do ... while`-Schleife können Sie eigentlich bereits alle Aufgaben umsetzen, die den Einsatz einer Schleife erfordern. Trotzdem gibt es in C# noch eine dritte Variante: die Zählschleife mit `for`. Sie kommt vor allem dann zum Einsatz, wenn Sie bereits vor dem Ausführen der Schleife wissen, wie oft die Anweisungen wiederholt werden müssen.

Das Grundprinzip einer `for`-Schleife unterscheidet sich nicht wesentlich von den beiden anderen Schleifenformen. Die Schleife wird ebenfalls so lange ausgeführt, bis eine Bedingung falsch ist. Die Syntax ist allerdings wesentlich kompakter. Ein Beispiel:

```
for (i = 0; i <= 10; i++)
    Anweisung;
```

Hinter dem Schlüsselwort `for` stehen in Klammern drei Ausdrücke, die das Verhalten der Schleife steuern.

Der **erste Ausdruck** `i = 0`; ist der **Vorlauf**. Er wird einmal vor Beginn der Schleife ausgeführt. In unserem Beispiel wird hier die Schleifenvariable `i` mit dem Wert 0 initialisiert.

Der **zweite Ausdruck** `i <= 10`; ist die **Bedingung** oder der **Testausdruck**. Er wird vor Beginn jedes Schleifendurchlaufs überprüft. Ergibt der Ausdruck falsch, wird die Ausführung der Schleife abgebrochen. In unserem Beispiel wird die Schleife also so lange ausgeführt, bis `i` den Wert 11 hat.

Der **dritte Ausdruck** `i++` wird als **Nachlauf** bezeichnet. Er wird am Ende jedes Schleifendurchlaufs ausgeführt. In unserem Beispiel wird die Schleifenvariable `i` also bei jedem Durchlauf um den Wert 1 erhöht.



Bitte beachten Sie:

Die drei Ausdrücke müssen jeweils durch ein Semikolon voneinander getrennt werden.

Die Anweisung, die die Schleife ausführen soll, steht erst **hinter** diesen drei Ausdrücken. Dabei können Sie – wie gewohnt – auch wieder mehrere Anweisungen in einem Anweisungsblock zusammenfassen.

Grafisch lässt sich die `for`-Schleife so darstellen:

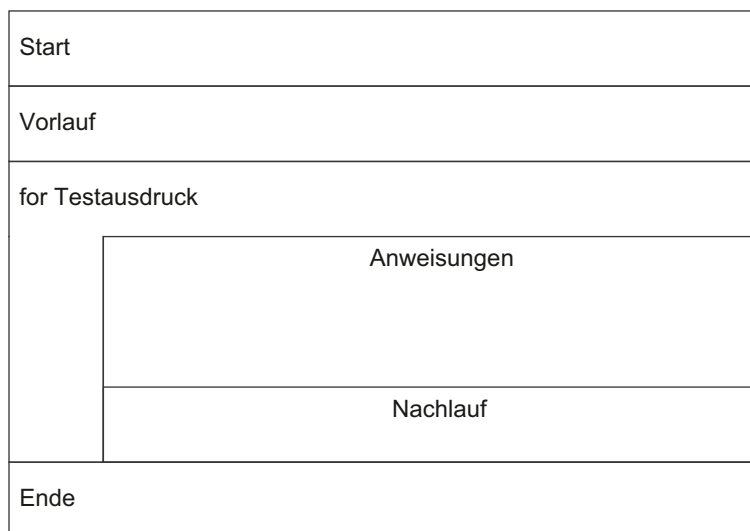


Abb. 3.3: Die `for`-Schleife



Bitte beachten Sie:

Die drei Ausdrücke in einer `for`-Schleife werden nicht gleichberechtigt abgearbeitet – auch wenn sie alle zusammen in einer Zeile stehen.

Der Vorlauf wird lediglich **einmal** bearbeitet – zu Beginn der Schleife. Bei der eigentlichen Ausführung der Schleife wird er **nicht** wiederholt.

Der Testausdruck wird **vor** jedem Schleifendurchlauf überprüft.

Der Nachlauf wird **nach** den Anweisungen der Schleife durchgeführt – obwohl er im Quelltext **vor** den Anweisungen steht.

Schauen wir uns die `for`-Schleife jetzt in einem C#-Programm an. Es gibt die Zahlen von 0 bis 10 auf dem Bildschirm aus:

```
/* #####
for-Schleife
##### */

using System;

namespace Cshp03d_03_05
{
    class Program
    {
        static void Main(string[] args)
        {
            int i;

            for(i = 0; i <= 10; i++)
                //bitte in einer Zeile eingeben
                Console.WriteLine("Die Variable hat jetzt den Wert
                {0}",i);
        }
    }
}
```

Code 3.5: `for`-Schleife

Dieser Code hat exakt dieselbe Wirkung wie Code 3.1. Damit Sie die beiden Codes leichter vergleichen können, drucken wir Code 3.1 hier noch einmal mit einigen Kommentaren ab und ändern außerdem den Namen der Schleifenvariablen in `i`.

```
/* #####
Von 0 bis 10 mit while
##### */

using System;

namespace Cshp03d_03_06
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;          //der Vorlauf
            while (i <= 10) //der Testausdruck
            {
                //bitte in einer Zeile eingeben
                Console.WriteLine("Die Variable hat jetzt den Wert
                {0}",i);
                i++;            //der Nachlauf
            }
        }
    }
}
```

Code 3.6: Von 0 bis 10 mit `while`

Der Vorlauf `i = 0;` steht im vorigen Code vor der Schleife und wird einmal vor Schleifenbeginn ausgeführt. Der Testausdruck `i <= 10` wird vor jedem Schleifendurchlauf überprüft, weil er im Schleifenkopf der `while`-Schleife steht. Der Nachlauf `i++;` wird am Ende nach jedem Schleifendurchlauf ausgeführt.

Im direkten Vergleich ist die Version mit `while` allerdings etwas länger und auch unübersichtlicher als die Version mit der `for`-Schleife. Denn im Code 3.5 wird ja allein durch einen kurzen Blick auf die Zeile mit `for` klar, wie die Schleifenvariable initialisiert wird, wie oft die Schleife läuft und wie die Veränderung der Schleifenvariable erfolgt. Beim Code 3.6 dagegen müssen Sie sich diese Informationen an mehreren Stellen zusammensuchen.

Code 3.5 lässt sich sogar noch etwas kompakter gestalten. Sie können die Schleifenvariable auch direkt im Vorlauf vereinbaren. Damit sparen Sie sich die Anweisung `int i;`. Der veränderte Code sieht dann so aus:

```
/* #####
Ein sehr kompakte for-Schleife
##### */

using System;

namespace Cshp03d_03_07
{
    class Program
    {
        static void Main(string[] args)
        {
            for(int i = 0; i <= 10; i++)
                //bitte in einer Zeile eingeben
                Console.WriteLine("Die Variable hat jetzt den Wert
                {0}", i);
        }
    }
}
```

Code 3.7: Eine sehr kompakte `for`-Schleife

Bitte beachten Sie aber, dass die Schleifenvariable `i` jetzt nur noch in der Schleife bekannt ist. Wenn Sie zum Beispiel versuchen, den aktuellen Wert von `i` außerhalb der Schleife auszugeben, erhalten Sie eine Fehlermeldung, dass der Name `i` im aktuellen Kontext nicht vorhanden ist. Warum diese Fehlermeldung erscheint, erfahren Sie später, wenn wir uns mit dem Gültigkeitsbereich von Variablen beschäftigen.

Sie können eine `for`-Schleife auch in die andere Richtung arbeiten lassen – sozusagen von oben nach unten. Das folgende Beispiel zählt von 100 so lange abwärts, bis die Schleifenvariable den Wert 0 hat:

```
for (int i = 100; i != 0; i--)
```

Bitte denken Sie daran, dass Schleifen so lange ausgeführt werden, **wie** die Bedingung **wahr** ist. Daher muss die Bedingung in dem Beispiel auch `i != 0` lauten. Wenn Sie hier nämlich die Bedingung `i == 0` verwenden, ist der Ausdruck direkt falsch, da ja `i` im Vorlauf auf den Wert 100 gesetzt wird. Die Anweisungen in der Schleife würden dann also gar nicht ausgeführt.

Bevor wir uns jetzt mit den Anweisungen `break` und `continue` in Schleifen beschäftigen, noch einige Hinweise.

Sie können die Bedingungen von Schleifen auch komplett mit Variablen oder mit einer Mischung aus Variablen und konstanten Werte formulieren – zum Beispiel:

```
while (a != i)
```

oder

```
while (a != i * 2)
```

Achten Sie aber in diesem Fall besonders sorgfältig darauf, dass die Bedingungen auch gültig sind. Andernfalls droht eine Endlosschleife.

Bei einer `for`-Schleife gehen viele Programmierer davon aus, dass die gesamte Steuerung der Schleife ausschließlich über die drei Ausdrücke hinter dem `for` erfolgt. Sie sollten daher die Schleifenvariable in einer `for`-Schleife nicht noch einmal innerhalb der Schleife verändern. Damit sorgen Sie nämlich unter Umständen bei anderen Programmierern, die Ihren Quelltext lesen, für sehr viel Verwirrung. Das folgende Beispiel ist zwar syntaktisch korrekt, gilt aber als schlechter Programmierstil:

```
for (int i = 0; i <= 10; i++)
{
    Console.WriteLine("Die Variable hat jetzt den Wert {0}", i);
    //hier wird i jetzt noch einmal in der Schleife verändert
    i = i + 2;
}
```

Beim Lesen der Zeile

```
for (int i = 0; i <= 10; i++)
```

werden die meisten Programmierer davon ausgehen, dass die Schleife exakt 11 Mal durchlaufen wird. Tatsächlich erfolgen aber nur 4 Durchläufe, da die Schleifenvariable `i` ja zusätzlich noch innerhalb der Schleife verändert wird. Sie sollten daher auf solche Konstruktionen bei `for`-Schleifen verzichten oder zumindest durch einen Kommentar im Quelltext auf das Verändern der Schleifenvariablen in der Schleife selbst hinweisen.

3.4 Die Anweisungen `break` und `continue` bei Schleifen

Neben der Steuerung von Schleifen über die Bedingung können Sie den Schleifenablauf auch noch durch die Anweisungen `break` und `continue` steuern.

Schauen wir uns zuerst die Anweisung `break` an.

3.4.1 break

Sie kennen diese Anweisung ja bereits von der `switch ... case`-Konstruktion. Dort markiert sie das Ende eines `case`- beziehungsweise `default`-Zweiges. Bei Schleifen wird `break` eingesetzt, um die Ausführung der Schleife komplett zu beenden. Nach einem `break` in einer Schleife werden also keine weiteren Durchläufe mehr durchgeführt und die Ausführung des Programms wird mit der nächsten Anweisung hinter dem Schleifenkörper fortgesetzt.

Sinnvoll ist die Anweisung `break` in einer Schleife allerdings nur, wenn Sie die Ausführung mit einer Bedingung koppeln. Sonst würde die Schleife ja sofort beim ersten Durchlauf abgebrochen und wäre damit überflüssig. Denkbar sind aber zum Beispiel Konstruktionen wie die folgende:

```
while (Ausdruck1)
{
    ...
    if (Ausdruck2)
        break;
    ...
}
```

Sobald `Ausdruck2` bei einem Schleifendurchlauf wahr wird, wird die Schleife über das `break` abgebrochen.

Sehen wir uns den Einsatz von `break` jetzt an einem praktischen Beispiel an. Im folgenden Code werden zehn Zahlen in einer Schleife eingelesen und zusammengezählt. Bei der Eingabe von 0 wird die Verarbeitung vorzeitig abgebrochen.

```
/* #####
break in einer Schleife
##### */

using System;

namespace Cshp03d_03_08
{
    class Program
    {
        static void Main(string[] args)
        {
            int summe, schleifenVariable, eingabe;
            //Initialisierung
            summe = 0;
            schleifenVariable = 1;

            while (schleifenVariable <= 10)
            {
                //bitte in einer Zeile eingeben
                Console.Write("Geben Sie die {0}. Zahl ein: ",
                    schleifenVariable);
                eingabe = Convert.ToInt32(Console.ReadLine());
```

```
        //wenn 0 eingegeben wurde, wird die Schleife abgebrochen
        if (eingabe == 0)
            break;
        //die Summe durch Zusammenrechnen bilden
        summe = eingabe + summe;
        schleifenVariable++;
    }

    Console.WriteLine("Das Einlesen ist beendet.");
    Console.WriteLine("Die Summe der Zahlen ist {0}.", summe);
}
}
```

Code 3.8: break in einer Schleife

Da dieser Code wieder etwas länger ist, schauen wir uns die wichtigsten Anweisungen der Reihe nach an.

Zuerst vereinbaren wir drei Variablen vom Typ `int` und initialisieren die Variablen `summe` und `schleifenVariable`. Die Schleifenvariable erhält dabei den Wert 1, damit wir später in der Schleife die Anzeige der einzugebenden Zahl synchron zum Schleifendurchlauf gestalten können. Die Variable `eingabe` dagegen müssen wir nicht initialisieren, da sie ja beim Einlesen einen Wert erhält.

Danach folgt dann die eigentliche Schleife. Hier geben wir zunächst einen Text aus und lesen dann eine Zahl ein. Das übernehmen die beiden Anweisungen

```
Console.Write("Geben Sie die {0}. Zahl ein: ",
    schleifenVariable);
eingabe = Convert.ToInt32(Console.ReadLine());
```

Anschließend überprüfen wir, ob `eingabe` den Wert 0 hat, und brechen gegebenenfalls die Schleife ab. Das erledigt die `if`-Anweisung

```
if (eingabe == 0)
    break;
```

Wenn `eingabe` nicht den Wert 0 hat, wird die `break`-Anweisung übersprungen und die Summe errechnet. Dazu addieren wir den Wert von `eingabe` und den aktuellen Wert von `summe`. Die entsprechende Anweisung

```
summe = eingabe + summe;
```

mag vielleicht ein wenig seltsam wirken, erfüllt aber beim genauen Hinsehen ihren Zweck. Denn es wird ja erst addiert und dann zugewiesen. Der Wert von `summe` rechts im Ausdruck entspricht damit immer der Summe aus dem letzten Durchlauf der Schleife. Dieser Wert wird mit der eingegebenen Zahl addiert und dann wieder in der Variablen `summe` abgelegt.

Tipp:

Falls Ihnen das Verfahren merkwürdig vorkommt, probieren Sie es einfach mit einem Bleistift und einem Blatt Papier aus. Führen Sie mehrere Schleifendurchläufe aus, und rechnen Sie dabei immer erst den Ausdruck rechts vom Zuweisungsoperator aus. Das Ergebnis weisen Sie dann wieder der Variablen `summe` zu.

Mit der letzten Anweisung in der Schleife schließlich wird die Schleifenvariable um den Wert 1 erhöht.

Am Ende des Codes werden dann eine Meldung und der Wert von `summe` ausgegeben.

Allerdings hat die `break`-Anweisung in Schleifen auch ihre Tücken. Denn Sie programmieren ja eine Art „Querausstieg“ aus der Schleife, der von einem sehr eiligen Leser möglicherweise übersehen wird. Etwas übersichtlicher ist es daher in solchen Fällen, die Ausführung der Schleife von einer weiteren Variablen abhängig zu machen, die nur markiert, ob die Schleife weiter ausgeführt werden soll oder nicht.



Variablen, die einen Zustand markieren, werden auch **Flag-Variablen** genannt. Wenn Sie nur zwischen zwei Zuständen unterscheiden müssen, können Sie für solche Variablen zum Beispiel den Typ `bool` benutzen.

Das Beispiel von oben könnte mit einer Flag-Variablen⁵ statt einer `break`-Anweisung folgendermaßen aussehen. Damit Sie die Änderungen schneller wiederfinden, haben wir sie fett markiert.

```
/* #####
Schleifenabbruch über eine Flag-Variable
##### */

using System;

namespace Cshp03d_03_09
{
    class Program
    {
        static void Main(string[] args)
        {
            int summe, schleifenVariable, eingabe;
            //die Flag-Variable vom Typ bool
            bool flagVariable = true;

            //Initialisierung
            summe = 0;
            schleifenVariable = 1;

            while ((schleifenVariable <= 10) &&(flagVariable == true))
            {
```

5. Wörtlich übersetzt bedeutet *flag* so viel wie „Flagge“. Bei der Programmierung ist damit aber ein eindeutiges Kennzeichen für die Unterscheidung von Zuständen gemeint.


```

        //bitte in einer Zeile eingeben
        Console.Write("Geben Sie die {0}. Zahl ein: ",
            schleifenVariable);
        eingabe = Convert.ToInt32(Console.ReadLine());
        //wenn 0 eingegeben wurde, wird flagVariable auf false
        //gesetzt
        //andernfalls wird gerechnet
        if (eingabe == 0)
            flagVariable = false;
        else
        {
            summe = eingabe + summe;
            schleifenVariable++;
        }
    }
    Console.WriteLine("Das Einlesen ist beendet.");
    Console.WriteLine("Die Summe der Zahlen ist {0}.", summe);
}
}
}

```

Code 3.9: Schleifenabbruch über eine Flag-Variable

Die Schleifensteuerung erfolgt jetzt durch eine logische UND-Verknüpfung der beiden Ausdrücke `schleifenVariable <= 10` und `flagVariable == true`. Die Schleife wird also nur noch dann ausgeführt, wenn beide Ausdrücke wahr sind. Damit wir auch für die Variable `flagVariable` einen eindeutigen Zustand haben, setzen wir sie bei der Vereinbarung auf den Wert `true`.

Hinweise:

Die Klammern um die beiden Ausdrücke dienen nur zur besseren Lesbarkeit. Sie können sie auch weglassen.

Den zweiten Ausdruck können Sie auch kompakter darstellen – nämlich einfach als `flagVariable`. Hier wird dann direkt der Wert von `flagVariable` verwendet. Die Variante ohne den ausdrücklichen Vergleich ist zwar kompakter, aber auch nicht mehr unbedingt beim flüchtigen Hinsehen zu verstehen.

Innerhalb der Schleife überprüfen wir dann wieder, ob `eingabe` den Wert 0 hat. Wenn das zutrifft, setzen wir `flagVariable` auf `false`. Damit wird die Schleife bei der nächsten Überprüfung des Ausdrucks im Schleifenkopf abgebrochen. Hat `eingabe` dagegen einen Wert ungleich 0, werden die beiden Anweisungen im `else`-Zweig ausgeführt. Dadurch wird sichergestellt, dass die Summe nur dann weitergeschrieben wird, wenn die Schleife nicht abgebrochen werden soll.

Probieren Sie den vorigen Code jetzt einmal aus. Sie werden sehen, er verhält sich genauso wie die Variante mit der `break`-Anweisung. Durch die logische Verknüpfung der beiden Ausdrücke in der Schleife wird jetzt aber sofort klar, dass die Ausführung der Schleife an **zwei Bedingungen** geknüpft ist. Das Programm ist zwar ein wenig länger, dafür aber übersichtlicher. Und das macht sich vor allem bei umfangreichen und komplexen Aufgaben sehr schnell bezahlt.

So viel zur Anweisung `break`. Kommen wir jetzt zu `continue`.

3.4.2 continue

Die Anweisung `continue`⁶ arbeitet ähnlich wie `break`, allerdings wird die Schleife nicht komplett abgebrochen, sondern es wird lediglich der aktuelle Schleifendurchlauf beendet.

Der Befehl `continue` sorgt also dafür, dass das Programm sofort zum Schleifenkopf bei einer `while`-Schleife beziehungsweise zum Schleifenfuß bei einer `do ... while`-Schleife springt. Dort wird dann die nächste Auswertung und danach eventuell der nächste Schleifendurchlauf ausgeführt.

Die allgemeine Syntax der `continue`-Anweisung sieht genauso aus wie bei `break`:

```
while (Ausdruck1)
{
    ...
    if (Ausdruck2)
        continue;
    ...
}
```

Sehen wir uns den Einsatz von `continue` an einem einfachen Beispiel an. Im folgenden Code wird das Quadrat einer eingegebenen Zahl in einer Schleife berechnet. Wenn die Zahl 0 eingegeben wird, springt das Programm direkt zum nächsten Schleifendurchlauf.

```
/* #####
continue in einer Schleife
##### */

using System;

namespace Cshp03d_03_10
{
    class Program
    {
        static void Main(string[] args)
        {
            int quadrat, schleifenVariable, eingabe;

            //Initialisierung
            quadrat = 0;
            schleifenVariable = 1;

            while (schleifenVariable <= 10)
            {
                //bitte in einer Zeile eingeben
                Console.Write("Geben Sie die {0}. Zahl ein: ",
                    schleifenVariable);
                eingabe = Convert.ToInt32(Console.ReadLine());
```

6. *Continue* bedeutet übersetzt so viel wie „setze fort“.

```

        //die Schleifenvariable muss jetzt vor
        //der Abfrage verändert werden
        schleifenVariable++;
        if (eingabe == 0)
            continue;
        //die folgenden Anweisungen werden nur ausgeführt,
        //wenn eingabe nicht 0 ist
        quadrat = eingabe * eingabe;
        //bitte in einer Zeile eingeben
        Console.WriteLine("Das Quadrat der Zahl ist
        {0}.", quadrat);
    }
    Console.WriteLine("Das Einlesen ist beendet.");
}
}
}

```

Code 3.10: continue in einer Schleife

Die `continue`-Anweisung finden Sie in dem Code in der `if`-Abfrage in der Schleife wieder.

```

        if (eingabe == 0)
            continue;

```

Wenn `eingabe` den Wert 0 hat, wird der aktuelle Durchlauf abgebrochen und der nächste Durchlauf gestartet. Das heißt, die Anweisungen nach der `if`-Abfrage werden dann nicht ausgeführt.

Bitte achten Sie bei `continue` sorgfältig auf das Verändern der Schleifenvariablen. In unserem Beispiel verändern wir die Schleifenvariable **vor** dem Aufruf der Anweisung `continue`. Damit ist sichergestellt, dass die Schleife insgesamt zehnmal durchlaufen wird. Ändern Sie die Schleifenvariable dagegen nach dem Aufruf von `continue`, können auch sehr viel mehr Durchläufe erfolgen. Denn dann wird die Schleifenvariable ja nur erhöht, wenn ein Wert ungleich 0 eingegeben wurde.

Da auch `continue`-Anweisungen zu recht unübersichtlichem Quelltext führen können, sollten Sie die Anweisung ebenfalls nur dann einsetzen, wenn Sie unbedingt müssen. In der Regel lassen sich nämlich mit einer `if`-Konstruktion die gleichen Ergebnisse erzielen. So könnten Sie im vorigen Code zum Beispiel die Berechnung nur dann ausführen lassen, wenn `eingabe` einen Wert ungleich 0 hat. Damit wäre das `continue` überflüssig. Der geänderte Code würde dann so aussehen:

```

/* #####
if-Abfrage statt continue
##### */

using System;

namespace Cshp03d_03_11
{
    class Program
    {
        static void Main(string[] args)
        {

```

```

int quadrat, schleifenVariable, eingabe;
//Initialisierung
quadrat = 0;
schleifenVariable = 1;
while (schleifenVariable <= 10)
{
    //bitte in einer Zeile eingeben
    Console.Write("Geben Sie die {0}. Zahl ein: ",
        schleifenVariable);
    eingabe = Convert.ToInt32(Console.ReadLine());

    //die Berechnungen werden nur ausgeführt, wenn ein
    //Wert ungleich 0 eingegeben wurde
    if (eingabe != 0)
    {
        quadrat = eingabe * eingabe;
        //bitte in einer Zeile eingeben
        Console.WriteLine("Das Quadrat der Zahl ist
            {0}.",quadrat);
    }
    schleifenVariable++;
}
Console.WriteLine("Das Einlesen ist beendet.");
}
}
}

```

Code 3.11: if-Konstruktion statt `continue`

Die Anweisungen `break` und `continue` können Sie grundsätzlich auch in `for`-Schleifen benutzen. Allerdings lässt sich dann der Schleifenablauf ebenfalls nicht mehr nur über die drei Anweisungen zu Beginn der Schleife ablesen. Deshalb sollten Sie die beiden Anweisungen in `for`-Schleifen nur sehr vorsichtig verwenden und ausführlich kommentieren.

Zum Abschluss dieses Kapitels wollen wir uns noch – wie versprochen – ansehen, wie Sie die Eingabe von ungültigen Daten abfangen können.

3.5 Exkurs: Abfangen von ungültigen Eingaben

Bisher mussten wir uns bei allen Programmen damit zufriedengeben, dass die Eingaben recht „wackelig“ sind. Wenn ein Anwender zum Beispiel statt – wie gefordert – einer Zahl ein Zeichen eingibt, stürzen die Programme ab, da die Konvertierung nicht möglich ist. Diese Abstürze lassen sich durch eine Schleife und die **Ausnahmebehandlung** – das **Exception Handling** – verhindern.

Das folgende Fragment liest zum Beispiel so lange Daten über die Tastatur ein, bis die Umwandlung über die Methode `Convert.ToInt32()` gelingt.

```

//gelingen ist vom Typ bool und muss mit false
//initialisiert werden
while (gelingen == false)
{

```

```
try
{
    eingabe = Convert.ToInt32(Console.ReadLine());
    gelungen = true;
}
catch (FormatException)
{
    Console.WriteLine("Ihre Eingabe war nicht gültig. Bitte
    wiederholen... ");
}
}
```

Code 3.12: Das Abfangen von Eingabefehlern

Entscheidend sind hier die Blöcke für `try` und `catch`. Im `try`-Block wird zunächst nur **versucht**, die Eingabe zu konvertieren. Wenn diese Konvertierung scheitert, werden die Verarbeitungen im `try`-Block abgebrochen und die Anweisungen aus dem `catch`-Block ausgeführt. Das heißt für das Beispiel: Die Schleifenvariable `gelungen` wird nur dann auf `true` gesetzt, wenn auch die Konvertierung im `try`-Block erfolgreich war. Scheitert die Konvertierung dagegen, behält die Schleifenvariable ihren Initialwert `false` und die Schleife wird wiederholt.

Damit wollen wir den Ausflug in die Ausnahmebehandlung an dieser Stelle auch schon wieder beenden. Mehr zu dem Thema erfahren Sie später.

Probieren Sie die Anweisungen aus dem vorigen Code aber jetzt schon einmal selbst aus. Ein vollständiges Beispiel finden Sie auch im Projekt `Cshp03d_03_12`.

Im nächsten Kapitel werden wir uns mit den Methoden beschäftigen.

Zusammenfassung

Mit Schleifen können Sie die Wiederholung von Anweisungen steuern.

C# kennt drei verschiedene Schleifentypen:

- die kopfgesteuerte Schleife mit `while`,
- die fußgesteuerte Schleife mit `do ... while` und
- die Zählschleife mit `for`.

Mit der Anweisung `break` können Sie die Ausführung einer Schleife komplett abbrechen. Die Anweisung `continue` bricht den aktuellen Durchlauf einer Schleife ab.

Aufgaben zur Selbstüberprüfung

- 3.1 Welcher wesentliche Unterschied besteht zwischen einer `while`- und einer `do ... while`-Schleife?

- 3.2 Die folgende Schleife soll von 10 bis 20 zählen. Welche Wirkung hat die Schleife tatsächlich? Warum?

```
int schleifenVariable = 10;
while (schleifenVariable <= 20)
{
    Console.WriteLine("{0}", schleifenVariable);
}
```

- 3.3 Ersetzen Sie im folgenden Quelltext die `break`-Anweisung in der `while`-Schleife durch eine andere Konstruktion. Die wesentliche Funktionalität soll dabei erhalten bleiben. Ein kleiner Tipp: Benutzen Sie eine Flag-Variable.

```
/* #####
break in einer Schleife
##### */

using System;

namespace LoesIII3
{
    class Program
    {
        static void Main(string[] args)
        {
            int i, k;
            i = 0;
            k = 0;
```

```
while (i <= 5)
{
    Console.Write("Geben Sie eine 1 zum Abbruch ein.");
    k = Convert.ToInt32(Console.ReadLine());
    if (k == 1)
        break;
    i++;
}

Console.WriteLine("Schleife beendet.");
}
```

4 Methoden

*In diesem Kapitel erfahren Sie, wie Sie Ihre Programme mit **Methoden** übersichtlicher und kompakter gestalten können. Sie lernen, wie Methoden in C# aufgebaut sind und wie Sie Werte aus Methoden zurückliefern beziehungsweise Werte an Methoden übergeben.*

4.1 Methoden in C#

Jetzt fragen Sie sich vielleicht: „Warum brauche ich überhaupt Methoden in einem Programm? Bisher lief doch alles wunderbar. Warum also die Sache unnötig komplizierter machen?“

Die Antworten auf diese Fragen sind einfach, wenn Sie sich die folgende Situation vorstellen:

Sie sollen ein Programm schreiben, in dem Sie an fünf verschiedenen Stellen zehn verschiedene Zahlen darauf überprüfen müssen, ob sie mit den Werten 5, 7, 12, 16 oder 21 übereinstimmen.

Kein Problem, sagen Sie? Sie erstellen einfach entsprechende `switch ... case`-Verzweigungen, die Sie dann in der benötigten Anzahl kopieren. Damit blähen Sie aber Ihren Quelltext unnötig auf, da Sie ja die gleiche Funktionalität mindestens fünfmal neu programmieren. Sehr viel effektiver und einfacher wäre es doch, den Quelltext für die Anweisungen nur einmal zu programmieren, aber beliebig oft nutzen zu können. Und genau das erreichen Sie mit Methoden.



Eine Methode wird einmal erstellt und kann dann im Quelltext beliebig oft aufgerufen werden. Methoden gehören immer zu einer Klasse.

In unserem Beispiel würden Sie also eine Methode schreiben, der Sie eine Zahl übergeben können. Die Methode gibt dann zurück, ob die Zahl mit den gewünschten Werten übereinstimmt oder nicht.

Damit sparen Sie sich nicht nur jede Menge Schreibarbeit, sondern halten auch den Quelltext sehr viel übersichtlicher. Denn der Code für die Überprüfung steht so nur einmal im Programm – und zwar in der Methode. An den Stellen, an denen Sie den Test durchführen wollen, rufen Sie diese Methode lediglich auf und übergeben dabei die Zahl, die getestet werden soll.

Der syntaktische Aufbau einer Methode in C# sieht so aus:

```
static Rückgabety p Name (Parametertypen)
{
    Anweisung1;
    Anweisung2;
    Anweisung...;
}
```


Die erste Zeile

```
static Rückgabetypp Name (Parametertypen)
```

bildet den **Kopf**. Er besteht aus dem Schlüsselwort `static`, dem **Rückgabetypp**, dem **Namen** der Methode und der Angabe der **Parameter**, die an die Methode übergeben werden. Die Parameter werden durch runde Klammern umfasst.

Hinweise:

Was es mit dem Rückgabetypp und den Parametern genau auf sich hat, erfahren Sie im weiteren Verlauf dieses Kapitels.

Bei Methoden, die mit dem Schlüsselwort `static`⁷ vereinbart werden, handelt es sich um eine spezielle Art von Methoden – die **Klassenmethoden**. Es gibt auch Methoden in Klassen, die ohne das Schlüsselwort `static` vereinbart werden. Damit werden wir uns im weiteren Verlauf des Lehrgangs bei der Objektorientierung beschäftigen. Hier gehen wir immer davon aus, dass die Methoden zu der Klasse gehören, die unser eigentliches Programm darstellt.

Für die Namensvergabe an Methoden gelten dieselben Regeln wie für die Namensvergabe an andere Bezeichner. Da Sie diese Regeln bereits kennen, wollen wir hier nur die wichtigsten Sachen noch einmal im Schnelldurchgang vorstellen:

- Methodennamen müssen mit einem Buchstaben oder einem `_` beginnen und dürfen ansonsten nur Buchstaben, Ziffern oder den Unterstrich `_` enthalten. `Test1` wäre zum Beispiel ein möglicher Name für eine Methode, während `1test` nicht zulässig ist.
- Groß- und Kleinschreibung werden unterschieden. `Test`, `TEST`, `test` und `TeSt` bezeichnen also vier unterschiedliche Methoden.
- Methodennamen dürfen nicht identisch mit Schlüsselwörtern sein. Schlüsselwörter als Teile des Namens sind dagegen erlaubt. `usingTest` wäre zum Beispiel ein möglicher Methodename, während `using` oder `int` als Namen nicht zulässig sind.

Auf den Kopf folgt der **Methodenkörper**. Er besteht aus einem Anweisungsblock – also aus einer Folge von Anweisungen in geschweiften Klammern. Im Methodenkörper steht der Programmcode, der beim Aufrufen der Methode ausgeführt werden soll. Bei sehr kurzen Methoden kann der Methodenkörper auch lediglich aus einer einzigen Anweisung bestehen. Syntaktisch korrekt, aber unsinnig wäre auch eine Methode mit einem leeren Körper – also ganz ohne Anweisung und damit auch ohne jede Wirkung.

Eine sehr einfache Methode in C# könnte damit so aussehen:

```
static void Methode()
{
}
```

Der Rückgabetypp dieser Methode wird durch das Schlüsselwort `void` gekennzeichnet. Es bedeutet so viel wie „leer“. Damit ist aber nicht gemeint, dass die Methode keinen Inhalt hat, sondern dass sie keinen Wert zurückliefert.

7. *Static* bedeutet wörtlich übersetzt statisch – also unveränderlich.



Methoden, die keinen Wert zurückgeben, haben den Rückgabebetyp `void`.

Werte werden von der Methode `Methode()` ebenfalls nicht verarbeitet. Deshalb ist die Liste in den Klammern hinter dem Namen der Methode leer.

Eine ganz bestimmte C#-Methode haben Sie übrigens schon die ganze Zeit in Ihren Programmen eingesetzt – nämlich die Methode `Main()`. Diese Methode bildet den Startpunkt des Programms. Aus dem Methodenkörper von `Main()` können Sie dann auch andere, selbst geschriebene Methoden aufrufen. Wie das funktioniert, zeigen wir Ihnen gleich.



Der Start einer C#-Anwendung erfolgt mit dem automatischen Aufruf der `Main()`-Methode. Jede C#-Anwendung benötigt daher eine `Main()`-Methode.

Sehen wir uns nun an einem Beispiel an, wie Sie Aufgaben in mehrere Methoden zerlegen und die Methoden dann aufrufen. Der folgende Code gibt lediglich zwei Texte auf dem Bildschirm aus. Wir werden ihn gleich so umbauen, dass die Ausgabe über zwei Methoden erfolgt.

```
/* #####
Einfache Textausgabe
##### */

using System;

namespace Cshp03d_04_01
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("C# macht Spaß.");
            Console.WriteLine("Aber nicht immer.");
        }
    }
}
```

Code 4.1: Einfache Textausgabe

Die beiden Ausgaben lassen wir nun jeweils durch eine Methode ausführen. Diese Methoden nennen wir `Ausgabe1()` und `Ausgabe2()`. Sie benötigen keinen Rückgabewert und auch keine Parameter. Also setzen wir den Rückgabewert auf `void` und lassen die Liste der Parameter hinter dem Namen der Methode leer.

Hinweis:

Wir benutzen die beiden Methoden lediglich als Beispiel, das Sie sehr einfach nachvollziehen können. In der Praxis ergibt es wenig Sinn, feste Texte über eine Methode ausgeben zu lassen, da keine Änderungen möglich sind und die Methode daher häufig nur an einer bestimmten Stelle eingesetzt werden kann.

Der Code mit den beiden Methoden sieht dann so aus:

```
/* #####  
Einfache Textausgabe mit Methoden  
##### */  
  
using System;  
  
namespace Cshp03d_04_02  
{  
    class Program  
    {  
        //die erste Methode Ausgabel()  
        static void Ausgabel()  
        {  
            Console.WriteLine("C# macht Spaß.");  
        }  
  
        //die zweite Methode Ausgabe2()  
        static void Ausgabe2()  
        {  
            Console.WriteLine("Aber nicht immer.");  
        }  
  
        static void Main(string[] args)  
        {  
            //der Aufruf der ersten Methode  
            Ausgabel();  
  
            //der Aufruf der zweiten Methode  
            Ausgabe2();  
            Console.WriteLine("Das war es.");  
        }  
    }  
}
```

Code 4.2: Einfache Textausgabe mit Methoden

Zu Beginn des Codes werden die beiden Methoden `Ausgabel()` und `Ausgabe2()` vereinbart. Sie geben jeweils lediglich eine Zeile Text aus.

Bitte beachten Sie:

Eine Klassenmethode kann nicht innerhalb einer anderen Methode vereinbart werden – auch nicht innerhalb von `Main()`. Die Vereinbarung einer Klassenmethode muss daher außerhalb jeder anderen Methode erfolgen, aber innerhalb der Klasse.

Die Reihenfolge der Vereinbarung im Quelltext ist beliebig. Sie können also eine Methode auch erst nach der ersten Aufrufstelle vereinbaren. Wichtig ist lediglich, dass sie in der Klasse steht und so vom Compiler „gefunden“ werden kann.



Der Aufruf der beiden Methoden `Ausgabel()` und `Ausgabe2()` erfolgt dann in der Methode `Main()` durch den Namen der Methode, gefolgt von der Liste der **Argumente**.

Hinweis:

Argumente sind die konkreten Werte, die an eine Funktion beim Aufruf übergeben werden. Mehr zum Unterschied zwischen Parameter und Argument erfahren Sie, wenn wir uns mit der Übergabe von Werten an Funktionen beschäftigen.

In unserem Beispiel verarbeiten allerdings beide Methoden keine Argumente. Deshalb sind die Listen immer leer – sowohl bei der Vereinbarung als auch beim Aufruf.

**Bitte beachten Sie:**

Auch wenn keine Argumente beziehungsweise Parameter verarbeitet werden, ist die Angabe der Klammern immer zwingend erforderlich – sowohl bei der Vereinbarung als auch beim Aufruf einer Methode. Sie dürfen die Klammern nicht weglassen!

Durch den Aufruf werden die Anweisungen in der entsprechenden Methode abgearbeitet. Nach dem Durchlaufen der Methode kehrt das Programm automatisch zurück an die Stelle, von der die Methode aufgerufen wurde, und setzt die Verarbeitung mit der folgenden Anweisung fort.

In unserem Beispiel wird also zuerst die Methode `Ausgabe1()` ausgeführt und danach die Methode `Ausgabe2()`. Nach dem Ausführen der Methode `Ausgabe2()` wird dann die letzte Zeile im Code abgearbeitet.

Grafisch lässt sich der Ablauf so darstellen:

Start von Main()
Aufruf von <code>Ausgabe1()</code> Ausgabe "C# macht Spaß."
Aufruf von <code>Ausgabe2()</code> Ausgabe "Aber nicht immer."
Ausgabe "Das war es."
Ende von Main()

Abb. 4.1: Der Programmablauf für Code 4.2

So viel zum Aufbau und zum Aufruf von Methoden. Schauen wir uns jetzt an, wie Sie Werte aus einer Methode zurückgeben lassen.

4.2 Methoden mit Rückgabewert

Durch den Rückgabewert können Sie das Ergebnis einer Methode weiterverarbeiten – zum Beispiel in der Methode `Main()`.

Die Rückgabe eines Wertes aus einer Methode erfolgt mit dem Schlüsselwort `return`. Hinter `return` wird der Wert angegeben, den die Methode liefern soll.



Zusätzlich müssen Sie den Typ der Rückgabe im Methodenkopf angeben. Dabei können Sie alle Typen verwenden, die Sie bereits von den Variablen kennen.

Die Angabe des Rückgabetyps ist zwingend erforderlich – auch dann, wenn die Methode keinen Wert liefert. In diesem Fall müssen Sie `void` verwenden.



Die folgende Methode `Rueckgabe()` liefert zum Beispiel den Wert 100 als `int`-Typ zurück.

```
static int Rueckgabe()
{
    return 100;
}
```

Bitte beachten Sie, dass eine Methode nach einem `return` sofort beendet wird. Alle weiteren Anweisungen, die eventuell noch folgen, werden nie ausgeführt. Im folgenden Beispiel würde also die Ausgabe nie auf dem Bildschirm erscheinen.

```
static int Rueckgabe()
{
    return 100;
    Console.WriteLine("Der Wert 100 wurde zurückgegeben.");
}
```

Daher meldet sich der Compiler bei solchen Konstruktionen auch mit einer Warnung, dass unerreichbarer Code entdeckt wurde.

Schauen wir uns die Methoden mit Rückgabewert jetzt an einem C#-Programm an. Das folgende Beispiel arbeitet mit zwei Methoden `Rueckgabe100()` und `Rueckgabe10()`, die den Wert 100 beziehungsweise 10 als `int`-Typ zurückgeben.

```
/* #####
Zwei Methoden mit Rückgabe
##### */

using System;

namespace Cshp03d_04_03
{
```

```
class Program
{
    //die Methode Rueckgabe100() vom Typ int
    static int Rueckgabe100()
    {
        return 100;
    }

    //die Methode Rueckgabe10() vom Typ int
    static int Rueckgabe10()
    {
        return 10;
    }

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

        //Aufruf der beiden Methoden in einer Ausgabe
        //bitte jeweils in einer Zeile eingeben
        Console.WriteLine("Die Methode Rueckgabe100() liefert den
Wert {0}.",Rueckgabe100());
        Console.WriteLine("Die Methode rueckgabe10() liefert den
Wert {0}.",Rueckgabe10());

        //Berechnungen gehen auch
        ergebnis = Rueckgabe100() + Rueckgabe10();
        Console.WriteLine("Das Ergebnis ist {0}.",ergebnis);

        //in Bedingungen kann eine Methode auch eingesetzt werden
        Console.Write("Geben Sie eine Zahl ein: ");
        ergebnis = Convert.ToInt32(Console.ReadLine());
        if (ergebnis < Rueckgabe100())
            Console.WriteLine("Ihre Eingabe war kleiner als 100.");
    }
}
```

Code 4.3: Zwei Methoden mit Rückgabewert

In der Methode `Main()` lassen wir zunächst einfach den Rückgabewert der beiden Methoden ausgeben. Dazu setzen Sie den Namen der Methode gefolgt von den leeren Klammern an die Stelle, an der sonst immer der Variablenname stand.

In der Anweisung

```
ergebnis = Rueckgabe100() + Rueckgabe10();
```

addieren wir dann die beiden Rückgabewerte und weisen das Ergebnis der Variablen `ergebnis` zu. Gerechnet wird hier also $100 + 10$.

In der `if`-Abfrage

```
if (ergebnis < Rueckgabe100())
```

schließlich verwenden wir den Rückgabewert der Methode `Rueckgabe100()` für einen Vergleich. Auch hier geben Sie einfach den Namen der Methode gefolgt von den Klammern an.

Sie sehen, Sie können die beiden Methoden an nahezu allen Stellen einsetzen, an denen Sie auch mit Zahlen oder Variablen vom Typ `int` arbeiten können – also bei Zuweisungen, bei Vergleichen, in Ausdrücken mit logischen Operatoren, in Verzweigungen und so weiter.

Noch einmal, weil es zu Beginn gerne vergessen wird:

Sie müssen in allen Fällen die Klammern hinter dem Namen der Methode angeben.



Immerhin meldet der Compiler in vielen Fällen einen Fehler, wenn Sie zum Beispiel eine Methode in einem Ausdruck verwenden wollen und dabei die Klammern vergessen.

Wenn Sie mit Methoden arbeiten, die Werte zurückliefern, müssen Sie darauf achten, dass die Methode auch wirklich **in jedem Fall** mit einem `return` beendet wird. Das heißt, die Anweisung `return` muss auch tatsächlich ausgeführt werden. Es reicht nicht, dass die Anweisung einfach im Quelltext steht.

Eine Methode, die einen Wert liefern soll, aber nicht mit `return` beendet wird, wird vom Compiler nicht akzeptiert.



Schauen wir uns dazu ein Beispiel an. Der folgende Code soll in der Methode `Eingabe()` einen Wert über die Tastatur einlesen und ihn an `Main()` zurückliefern. Die Rückgabe erfolgt allerdings nur dann, wenn der eingelesene Wert kleiner oder gleich 20 ist.

```
/* #####
Undefinierte Rückgabe aus einer Methode
Das Beispiel lässt sich nicht übersetzen
##### */

using System;

namespace Cshp03d_04_04
{
    class Program
    {
        static int Eingabe()
        {
            int einVariable;

            Console.Write("Geben Sie eine Zahl ein: ");
            einVariable = Convert.ToInt32(Console.ReadLine());
        }
    }
}
```

```

        if (einVariable <= 20)
            return einVariable;
    }

    static void Main(string[] args)
    {
        //bitte in einer Zeile eingeben
        Console.WriteLine("Die Methode Eingabe() liefert den Wert
        {0}.",Eingabe());
    }
}

```

Code 4.4: undefinierte Rückgabe aus einer Methode (der Code lässt sich nicht übersetzen)

Hier beschwert sich der Compiler zu Recht, dass in der Methode `Eingabe()` nicht alle Codepfade einen Wert zurückliefern – obwohl eine `return`-Anweisung vorhanden ist. Die wird allerdings nicht in jedem Fall auch tatsächlich ausgeführt.

Sie müssen außerdem darauf achten, dass der Rückgabotyp der Methode zu dem Typ des Wertes passt, der mit `return` verarbeitet wird. Schauen wir uns auch dazu ein Beispiel an. Wir überarbeiten die Methode `Eingabe()` aus dem vorigen Code einmal so, dass sie einen `char`-Typ zurückliefert. Den Typ der Variablen `einVariable` in der Methode lassen wir unverändert.

```

/* #####
Unterschiedliche Typen in einer Methode
Das Beispiel lässt sich ebenfalls nicht übersetzen
##### */

using System;

namespace Cshp03d_04_05
{
    class Program
    {
        static char Eingabe()
        {
            int einVariable;

            Console.Write("Geben Sie eine Zahl ein: ");
            einVariable = Convert.ToInt32(Console.ReadLine());
            return einVariable;
        }

        static void Main(string[] args)
        {
            //bitte in einer Zeile eingeben
            Console.WriteLine("Die Methode Eingabe() liefert den Wert
            {0}.", Eingabe());
        }
    }
}

```

Code 4.5: Unterschiedliche Typen (der Code lässt sich ebenfalls nicht übersetzen)

Hier beschwert sich der Compiler jetzt bei der `return`-Anweisung in der Methode, dass er den Typ `int` nicht in einen `char` konvertieren kann.

Hinweis:

Ob Sie den Wert hinter `return` in Klammern setzen oder nicht, ist bei der Rückgabe eines einzelnen Wertes übrigens beliebig. Häufig werden die Klammern zur besseren Lesbarkeit gesetzt.

Mit Standardtechniken kann eine Methode in C# immer nur einen einzigen Wert zurückgeben. Sie können also zum Beispiel in einer Methode nicht zwei Werte einlesen und dann ohne Weiteres beide Werte zurückgeben. Der folgende Code wird daher ebenfalls nicht übersetzt.

```
/* #####
Das klappt nicht!
##### */

using System;

namespace Cshp03d_04_06
{
    class Program
    {
        static int Eingabe()
        {
            int einVariable1, einVariable2;

            Console.Write("Geben Sie Zahl 1 ein: ");
            einVariable1 = Convert.ToInt32(Console.ReadLine());

            Console.Write("Geben Sie Zahl 2 ein: ");
            einVariable2 = Convert.ToInt32(Console.ReadLine());
            return (einVariable1, einVariable2);
        }

        static void Main(string[] args)
        {
            //bitte in einer Zeile eingeben
            Console.WriteLine("Die Methode Eingabe() liefert den Wert {0}.",Eingabe());
        }
    }
}
```

Code 4.6: Das klappt nicht!

Hier beschwert sich der Compiler ebenfalls wieder über die Rückgabe aus der Methode.

Damit die Rückgabe mehrerer Werte möglich ist, müssen Sie ein Tupel verwenden. Der geänderte Code sieht dann so aus:

```
/* #####
Rückgabe mehrerer Werte über ein Tupel
##### */

using System;

namespace Cshp03d_04_07
{
    class Program
    {
        //die Methode gibt ein Tupel zurück
        static (int, int) Eingabe()
        {
            int einVariable1, einVariable2;

            Console.Write("Geben Sie Zahl 1 ein: ");
            einVariable1 = Convert.ToInt32(Console.ReadLine());

            Console.Write("Geben Sie Zahl 2 ein: ");
            einVariable2 = Convert.ToInt32(Console.ReadLine());

            return (einVariable1, einVariable2);
        }

        static void Main(string[] args)
        {
            //die Werte aus der Methode zuweisen
            //die Zuweisung erfolgt auf ein Tupel
            (int wert1, int wert2) = Eingabe();
            //bitte in einer Zeile eingeben
            Console.WriteLine("Die Methode Eingabe() liefert
            die Werte {0} und {1}.", wert1, wert2);
        }
    }
}
```

Code 4.7: Rückgabe mehrerer Werte über ein Tupel

Im Kopf der Methode wird als Rückgabewert jetzt ein Tupel vereinbart.



Ein Tupel ist eine geordnete Menge von Werten.

Dazu geben Sie einfach die Typen für die Werte an, die Sie zurückliefern wollen, und trennen Sie durch ein Komma. Die Liste setzen Sie dabei in runde Klammern.

Der Kopf

```
static (int, int) Eingabe()
```

legt also fest, dass unsere Methode `Eingabe()` zwei `int`-Werte zurückliefert.

Bei der Rückgabe geben Sie dann die beiden Werte hintereinander in runden Klammern an und trennen Sie ebenfalls durch Kommas.

Bei der Rückgabe eines Tupels sind die runden Klammern bei `return` zwingend erforderlich.



Interessant ist dann noch einmal die Zuweisung der Ergebnisse in der Methode `Main()`. Sie erfolgt durch die Anweisung

```
(int wert1, int wert2) = Eingabe();
```

In den runden Klammern stehen dabei die Typen und Bezeichner für die Variablen, die die Werte speichern sollen. Die Angaben trennen Sie dabei wieder durch ein Komma. Dann erfolgt die Zuweisung durch den Operator `=` und den Aufruf der Methode.

Bei der Ausgabe gibt es keine Besonderheiten. Hier greifen wir auf die beiden Variablen zu, denen wir die Rückgabe der Methode zugewiesen haben und geben die Werte aus.

4.3 Methoden mit Argumenten

Sonderlich nützlich waren unsere Methoden bisher noch nicht. Denn alles, was wir in den vorigen Beispielen mit den Methoden umgesetzt haben, lässt sich genauso gut auch ohne Methode erledigen – sogar mit weniger Schreibaufwand.

Richtig spannend werden Methoden nämlich erst, wenn Sie sie mit Werten füttern. Dazu übergeben Sie beim Aufruf der Methode **Argumente**.

Die Begriffe **Argument** und **Parameter** werden oft nicht eindeutig unterschieden. Streng genommen sind **Parameter** die Angaben in den runden Klammern im Methodenkopf. **Argumente** dagegen sind die Werte, die beim Aufruf der Methode in den runden Klammern angegeben werden. Häufig wird der Begriff **Parameter** aber für beides verwendet – also sowohl für die eigentlichen Parameter als auch für die Argumente.



Damit Sie Argumente an eine Methode übergeben können, sind zwei Erweiterungen erforderlich:

- 1) Sie müssen im **Methodenkopf** in den Klammern die Typen und die Namen der Parameter angeben.
- 2) Sie müssen die Werte – die Argumente – beim **Aufruf** der Methode in den Klammern angeben.

Schauen wir uns zuerst die Änderungen im Methodenkopf an einem Beispiel genauer an:

```
static int Quadrat(int zahl)
```

Vereinbart wird hier eine Methode `Quadrat()`, die einen Wert vom Typ `int` zurückliefert. Die Methode verarbeitet einen Parameter vom Typ `int` mit dem Namen `zahl`. Durch die Angabe in der Parameterliste kann die Variable `zahl` innerhalb der Methode

genauso benutzt werden wie jede andere innerhalb der Methode vereinbarte Variable auch. Der einzige Unterschied besteht darin, dass die Variable bereits mit einem Wert initialisiert ist, der beim Aufruf der Methode übergeben wird.

Außerhalb der Methode ist die Variable allerdings nicht bekannt. Das heißt, sie gilt ausschließlich für die Methode.



Eine Variable, die Sie als Parameter einer Methode verwenden, wird automatisch vereinbart. Sie müssen keine eigene Anweisung für die Vereinbarung benutzen.

Der Aufruf der Methode `Quadrat()` kann dann zum Beispiel so erfolgen:

```
Console.WriteLine("Das Quadrat der Zahl {0} ist {1}", 10,
    Quadrat(10));
```

Hier wird die Zahl 10 als Argument an die Methode `Quadrat()` übergeben. Möglich wäre aber auch ein Aufruf wie

```
Console.WriteLine("Das Quadrat der Zahl {0} ist {1}", a,
    Quadrat(a));
```

Hier wird der Wert der Variablen `a` als Argument an die Methode übergeben.

Grundsätzlich können Sie an eine Methode alles als Argument übergeben, was Sie auch dem Parameter im Methodenkopf zuweisen können. Für unser Beispiel wären also auch komplexe Ausdrücke wie `a * b / c` möglich. An die Methode wird dann der Wert des Ausdrucks als Argument übergeben.



Lassen Sie sich durch die verschiedenen Bezeichner des Parameters im Methodenkopf und des Arguments beim Aufruf nicht verwirren.

Der Bezeichner des Parameters im Methodenkopf ist eine Art Platzhalter für das Argument. Sie müssen also beim Aufruf nicht den Bezeichner des Parameters verwenden, sondern können für den Platzhalter beliebige Werte und Ausdrücke einsetzen. Sie müssen lediglich darauf achten, dass der Typ des Arguments zum Typ des Parameters passt.

Sehen wir uns jetzt die Arbeit mit Argumenten an einem praktischen Beispiel an. Im folgenden Code wird das Quadrat einer Zahl über eine Methode berechnet und das Ergebnis dann in der Methode `Main()` ausgegeben.

```
/* #####
Eine Methode mit Argument
##### */

using System;

namespace Cshp03d_04_08
{
    class Program
    {
```

```

static int Quadrat(int zahl)
{
    int ergebnis;

    //zahl ist aus der Parameterliste bekannt
    ergebnis = zahl * zahl;
    return ergebnis;
}

static void Main(string[] args)
{
    int einVariable;

    Console.Write("Geben Sie eine Zahl ein: ");
    einVariable = Convert.ToInt32(Console.ReadLine());

    //einVariable wird als Argument an die Methode übergeben
    //bitte in einer Zeile eingeben
    Console.WriteLine("Das Quadrat der Zahl {0} ist {1}.",
        einVariable, Quadrat(einVariable));
}
}

```

Code 4.8: Eine Methode mit Argument

Das Programm liest zunächst einen Wert in die Variable `einVariable` ein. Dieser Wert wird dann beim Aufruf an die Methode in Klammern als Argument übergeben.

In der Methode selbst findet sich der Wert von `einVariable` in der Variablen `zahl` wieder. Diese Variable wurde im Methodenkopf in der Parameterliste vereinbart.

Nach der Berechnung liefert die Methode das Ergebnis über die `return`-Anweisung zurück. Abschließend erfolgt die Ausgabe des zurückgegebenen Wertes über `Console.WriteLine()`.

Bitte beachten Sie:

Bei der Übergabe der Argumente müssen Sie darauf achten, dass der Typ des Arguments zum Typ des Parameters passt. Wenn Sie in dem Code oben zum Beispiel den Typ der Variablen `einVariable` in `float` ändern, beschwert sich der Compiler, dass er das Argument für die Methode `Quadrat()` nicht konvertieren kann.



Schauen wir uns nun noch an, wie Sie mehrere Argumente an eine Methode übergeben.

Dazu geben Sie einfach die Argumente hintereinander in der gewünschten Reihenfolge in den Klammern beim Aufruf an und trennen sie durch Kommas. Außerdem müssen Sie auch noch die Parameterliste im Methodenkopf so erweitern, dass mehrere Werte verarbeitet werden können.

Im folgenden Beispiel sehen Sie sowohl den Kopf als auch den Aufruf einer Methode mit zwei Parametern beziehungsweise Argumenten:

```

static int Summe(int x, int y) //Kopf
Summe(a,b)                    //Aufruf der Methode

```


Bitte beachten Sie:

Sie müssen in der Parameterliste für **jeden** Parameter den Typ einzeln angeben. Die verkürzte Form

```
static int Summe(int x, y)
```

wird vom Compiler nicht akzeptiert.

Die Anzahl der Argumente beim Aufruf und der Parameter bei der Vereinbarung muss übereinstimmen. Wenn Sie zum Beispiel an die Methode `Summe()` nur ein Argument übergeben, meldet der Compiler, dass die Methode für diese Anzahl Argumente nicht verfügbar ist.

Eine ähnliche Meldung erscheint auch dann, wenn Sie mehr Argumente angeben, als die Methode verarbeiten kann.

Im praktischen Einsatz finden Sie die Methode `Summe()` im folgenden Code. Das Programm addiert zwei Zahlen, die Sie eingeben, und gibt das Ergebnis auf dem Bildschirm aus.

```
/* #####
Eine Methode mit mehreren Argumenten
##### */

using System;

namespace Cshp03d_04_09
{
    class Program
    {
        static int Summe(int x, int y)
        {
            return (x + y);
        }

        static void Main(string[] args)
        {
            int einVariable1, einVariable2;

            Console.Write("Geben Sie die erste Zahl ein: ");
            einVariable1 = Convert.ToInt32(Console.ReadLine());

            Console.Write("Geben Sie die zweite Zahl ein: ");
            einVariable2 = Convert.ToInt32(Console.ReadLine());
            //bitte in einer Zeile eingeben
            Console.WriteLine("Die Summe der beiden Zahlen ist
            {0}", Summe(einVariable1, einVariable2));
        }
    }
}
```

Code 4.9: Eine Methode mit zwei Argumenten

Die Schreibweise

```
return (x + y);
```

für die Rückgabe des Wertes aus der Methode ist die verkürzte Version von

```
int ergebnis;
ergebnis = x + y;
return ergebnis;
```

Da $x + y$ ein berechenbarer Ausdruck ist und einen `int`-Wert liefert, können wir ihn direkt hinter `return` angeben und uns eine eigene Variable für die Rückgabe sparen.

Experimentieren Sie mit dem vorigen Code ein wenig. Versuchen Sie zum Beispiel einmal, die Summe aus vier oder fünf Werten in der Methode zu berechnen.

4.4 Tipps zum Arbeiten mit Methoden

Der Editor von Visual Studio stellt Ihnen einige Funktionen zur Verfügung, die die Arbeit mit Methoden erleichtern können – vor allem dann, wenn Sie später einmal ein Programm mit sehr vielen Methoden erstellen.

So finden Sie zum Beispiel links neben jedem Methodenkopf ein Symbol mit einem Minus-Zeichen und einer Linie. Durch einen Mausklick auf dieses Symbol können Sie den gesamten Körper der jeweiligen Methode ausblenden und so die Anzeige des Quelltextes sehr kompakt halten.

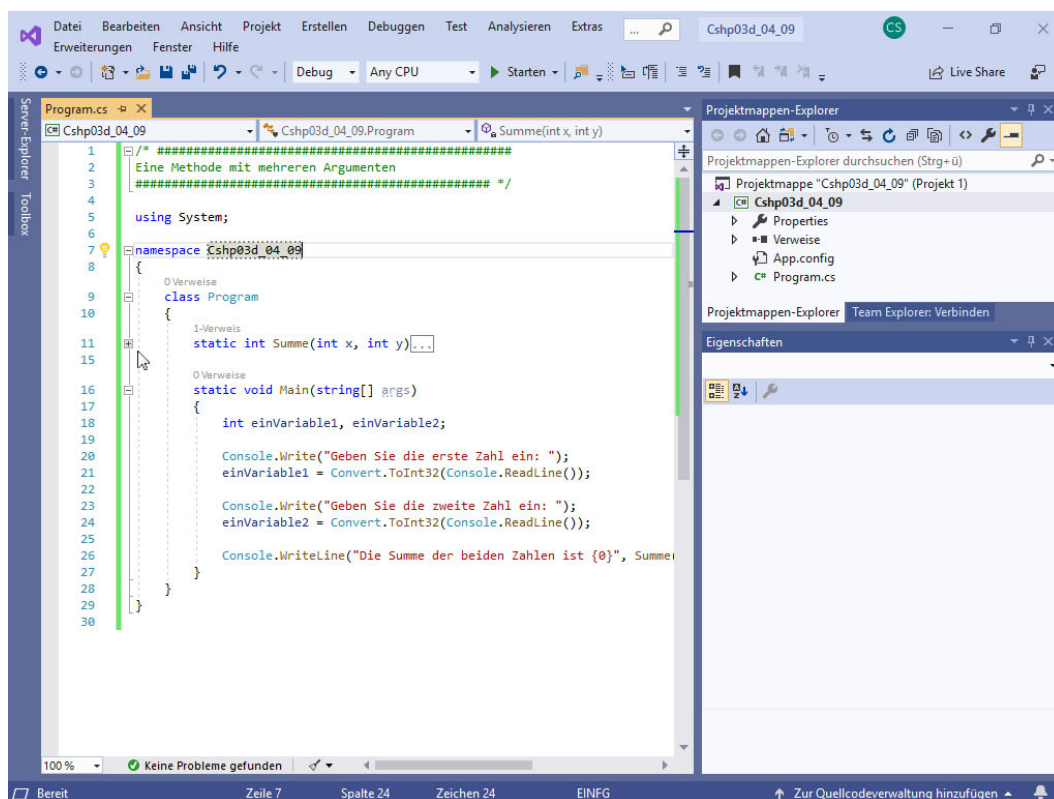


Abb. 4.2: Ein ausgeblendeter Methodenkörper (links in der Mitte der Abbildung am Mauszeiger)

Wenn Sie den Quelltext der Methode wieder anzeigen möchten, klicken Sie noch einmal auf das Symbol. Alternativ können Sie den Mauszeiger auch auf den Kasten hinter dem Methodenkopf stellen. Dann erscheint der Quelltext der Methode als Quickinfo.

Über das Kombinationsfeld rechts oben im Editor können Sie sehr schnell zwischen verschiedenen Methoden im Quelltext hin und her springen. Dazu öffnen Sie das Kombinationsfeld und klicken dann in der Liste auf den Eintrag der gewünschten Methode. Außerdem können Sie in der Liste auch ablesen, welche Argumente eine Methode erwartet.

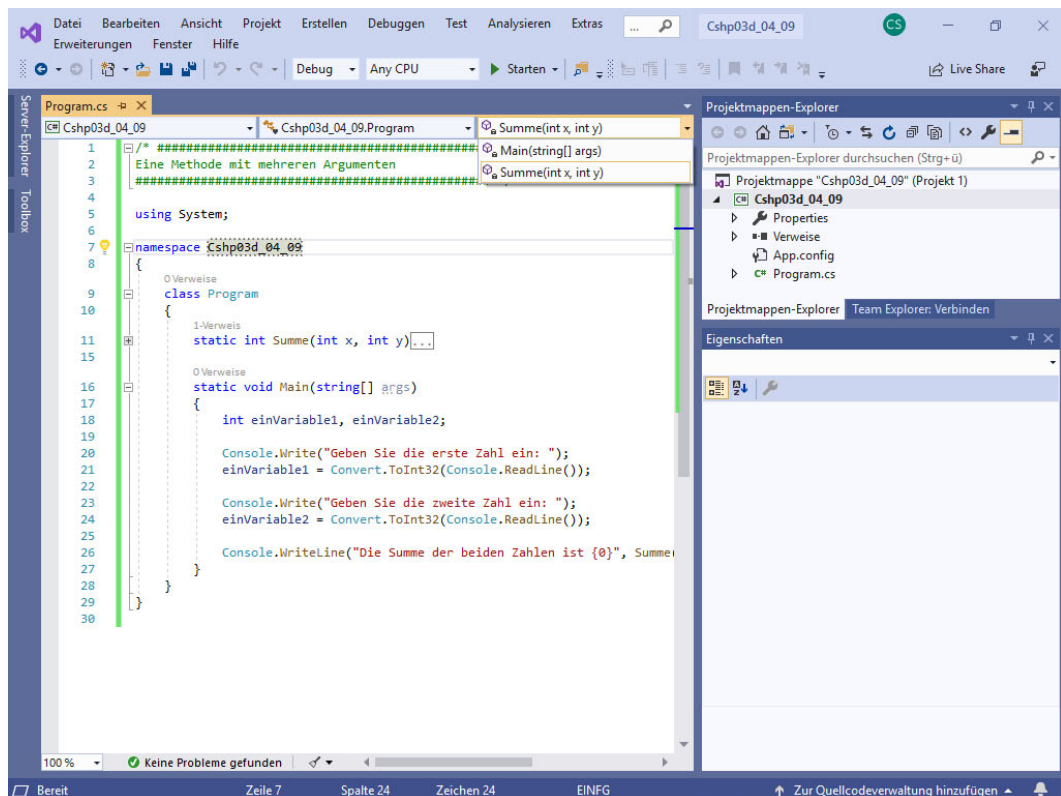
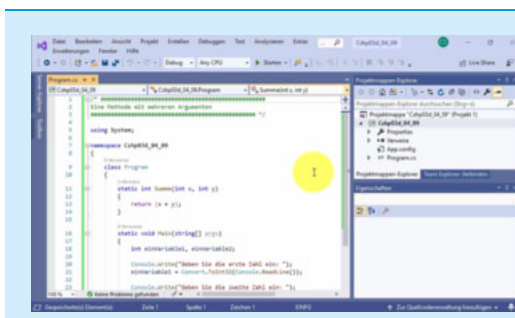


Abb. 4.3: Die Liste der Methoden (oben in der Mitte der Abbildung)

Informationen zu den Argumenten werden übrigens auch dann angezeigt, wenn Sie den Namen der Methode und die Klammer (im Quelltext eingeben. Dann erscheinen die Parameterliste und auch der Rückgabetyp in einer Quickinfo.



In diesem Video stellen wir Ihnen die Hilfen zum Arbeiten mit Methoden im praktischen Einsatz vor.

www.dfz.media/3cuu12

Video 4.1: Hilfen zum Arbeiten mit Methoden

Abschließend noch ein Hinweis:

Lassen Sie sich von den Begriffen Parameter und Argument nicht verwirren. Wichtig ist vor allem, dass Sie an eine Methode Werte übergeben können. Die Anzahl und der Typ dieser Werte werden bei der Vereinbarung der Methode angegeben und dienen dort quasi als Platzhalter. Beim Aufruf geben Sie dann die konkreten Werte an, die an die Methode übergeben werden sollen. Dabei müssen Typ und Anzahl mit der Liste im Kopf der Methode übereinstimmen. Der Bezeichner dagegen ist beliebig.

Zusammenfassung

Eine Methode wird einmal erstellt und kann dann beliebig oft im Quelltext aufgerufen werden.

Eine Methode besteht aus dem Kopf und dem Körper. Der Kopf enthält den Typ des Rückgabewertes, den Namen der Methode und eine Liste der Parameter. Der Körper enthält die Anweisungen, die die Methode ausführen soll.

Der Aufruf einer Methode erfolgt durch den Namen der Methode gefolgt von einer Liste der Argumente.

Auch bei Methoden ohne Parameter müssen Sie die Klammern `()` angeben – sowohl bei der Vereinbarung als auch beim Aufruf.

Mit der Anweisung `return` geben Sie einen Wert aus einer Methode zurück. Der Rückgabewert steht hinter dem `return`.

Wenn Sie mehrere Parameter beziehungsweise Argumente verwenden, müssen Sie die einzelnen Werte durch Kommas trennen.

Aufgaben zur Selbstüberprüfung

- 4.1 Welchen Wert liefert die folgende Methode zurück? Sehen Sie sich bitte vor der Antwort den Kopf der Methode genau an.

```
Summe(int a, int b)
{
    return (a + b);
}
```

- 4.2 Wie viele Werte können Sie mit Standardtechniken aus einer Methode zurückliefern lassen? Wie können Sie mehrere Werte zurückgeben lassen?

- 4.3 Sehen Sie sich bitte den folgenden Methodenkopf an. Was stimmt nicht? Wie muss der Methodenkopf korrekt lauten?

```
char Buchstabe(char a, b)
```

- 4.4 Schreiben Sie ein kurzes Programm, das eine Methode `Produkt()` enthält, die zwei Zahlen `x` und `y` miteinander multipliziert und das Ergebnis zurückgibt. Überprüfen Sie die Funktionsweise der Methode `Produkt()` durch einen Aufruf aus der Methode `Main()`.

- 4.5 Schreiben Sie eine Methode `Quadrat()`, die das Quadrat einer Zahl `x` berechnet. Setzen Sie zum Berechnen aber nicht den Operator `*` ein, sondern rufen Sie in der Methode `Quadrat()` die Methode `Produkt()` aus der Aufgabe 4.4 auf. Schreiben Sie eine passende Methode `Main()`, um die korrekte Funktion zu überprüfen.

Schlussbetrachtung

Sie dürfen jetzt ein wenig stolz auf sich sein. Sie haben in diesem Studienheft einige schon recht komplexe Anweisungen kennengelernt und auch ausprobiert.

Sie können jetzt Programme erstellen, die flexibel auf Bedingungen reagieren, Anweisungen wiederholen und auch Anweisungen in Methoden mehrfach einsetzen, ohne sie erneut eingeben zu müssen.

Vielleicht kommt Ihnen das eine oder andere in diesem Studienheft noch etwas seltsam und fremd vor. Nehmen Sie sich dann die Zeit und arbeiten Sie die entsprechenden Kapitel noch einmal in aller Ruhe durch. Probieren Sie dabei auch die Beispielcodes aus und haben Sie keine Scheu, Änderungen an den Programmen vorzunehmen. Viele scheinbar schwierige Konstruktionen werden durch wiederholtes Ausprobieren sehr viel leichter verständlich.

Denken Sie daran: Nur Übung macht den Meister!

Und vor allem: Lassen Sie sich Zeit. Beim Programmieren geht es nicht um Geschwindigkeit, sondern vor allem um ein stabiles und korrekt funktionierendes Endergebnis. Denken Sie deshalb auch erst einmal in Ruhe nach, wie Sie überhaupt ein Problem mit einem Programm lösen wollen, und fangen Sie nicht an, „wild“ drauflos zu programmieren.

Christoph Siebeck

A. Lösungen der Aufgaben zur Selbstüberprüfung

Hier finden Sie die Lösungen zu den Aufgaben zur Selbstüberprüfung in den einzelnen Kapiteln. Bei offenen Aufgaben mit freien Formulierungen kommt es nicht auf eine wörtliche Übereinstimmung an, sondern auf den Inhalt. Entsprechen Ihre Ergebnisse nicht den Lösungen, wiederholen Sie bitte das entsprechende Kapitel und bearbeiten Sie die zugehörigen Aufgaben zur Selbstüberprüfung nach einer Pause erneut.

Kapitel 1

1.1 Wenn Sie zwei Zahlen auf Gleichheit testen wollen, benutzen Sie den Operator `==`. Mit dem Operator `>` können Sie prüfen, ob die Zahl links vom Operator größer ist als die Zahl rechts vom Operator. Alternativ können Sie auch den Operator `<` verwenden. Dann müssen allerdings die Operanden getauscht werden.

- 1.2
- a) wahr
 - b) falsch
 - c) wahr
 - d) falsch
 - e) wahr
 - f) wahr
 - g) falsch

1.3

A	B	!(A B)
falsch	falsch	wahr
falsch	wahr	falsch
wahr	falsch	falsch
wahr	wahr	falsch

Kapitel 2

2.1 Die Lösung könnte so aussehen:

```
/* #####
Lösung II.1
##### */

using System;

namespace LoesII1
{
    class Program
    {
        static void Main(string[] args)
        {
            int x, y;

            Console.Write("Geben Sie den Wert für x ein: ");
            x = Convert.ToInt32(Console.ReadLine());
```

```

        Console.WriteLine("Geben Sie den Wert für y ein: ");
        y = Convert.ToInt32(Console.ReadLine());

        if (x>y)
            Console.WriteLine("x ist größer als y.");
        else
            Console.WriteLine("y ist größer oder gleich x.");
    }
}

```

2.2 Die fehlenden Teile sind im folgenden Quelltext fett markiert.

```

/* #####
Lösung II.2
##### */

using System;

namespace LoesII2
{
    class Program
    {
        static void Main(string[] args)
        {
            int essenWahl;

            Console.WriteLine("Wählen Sie bitte ein Essen aus:
\n");
            Console.WriteLine("1 Jägerschnitzel mit Pommes");
            Console.WriteLine("2 Currywurst mit Pommes");
            Console.WriteLine("3 Bratwurst mit Brötchen\n");
            Console.Write("Was möchten Sie essen? ");

            essenWahl = Convert.ToInt32(Console.ReadLine());

            switch (essenWahl)
            {
                case 1:
                    Console.WriteLine("Sie haben ein Jägerschnitzel
mit Pommes gewählt.");
                    break;
                case 2:
                    Console.WriteLine("Sie haben eine Currywurst mit
Pommes gewählt.");
                    break;
                case 3:
                    Console.WriteLine("Sie haben eine Bratwurst mit
Brötchen gewählt.");
                    break;
            }
        }
    }
}

```

Kapitel 3

- 3.1 Bei der `while`-Schleife wird die Bedingung vor dem Schleifendurchlauf geprüft, bei der `do ... while`-Schleife erst nach dem Schleifendurchlauf. Daher wird die `do ... while`-Schleife auch in jedem Fall mindestens einmal ausgeführt.
- 3.2 Die Schleife gibt endlos die Zahl 10 aus. Es fehlt die Veränderung der Schleifenvariablen.
- 3.3 Die Lösung könnte so aussehen:

```
/* #####
Lösung III.3
##### */

using System;

namespace LoesIII3
{
    class Program
    {
        static void Main(string[] args)
        {
            int i, k;
            bool flagVariable = true;
            i = 0;
            k = 0;

            while ((i <= 5) && (flagVariable == true))
            {
                Console.WriteLine("Geben Sie eine 1 zum Abbruch ein.");
                k = Convert.ToInt32(Console.ReadLine());
                if (k == 1)
                    flagVariable = false;
                else
                    i++;
            }

            Console.WriteLine("Schleife beendet.");
        }
    }
}
```

Kapitel 4

- 4.1 Die Methode liefert keinen Wert zurück, da sich das Programm gar nicht übersetzen lässt. Es fehlt der Rückgabetypp für die Methode.
- 4.2 Eine Methode kann mit Standardtechniken nur einen Wert zurückliefern. Um mehrere Werte zurückzuliefern, müssen Sie ein Tupel verwenden.
- 4.3 In der Liste der Parameter fehlt die Typangabe für den zweiten Parameter. Korrekt wäre

```
char Buchstabe(char a, char b)
```

4.4 Die Lösung könnte so aussehen:

```

/* #####
Lösung IV.4
##### */

using System;

namespace LoesIV4
{
    class Program
    {
        static int Produkt(int x, int y)
        {
            return (x * y);
        }

        static void Main(string[] args)
        {
            int einVariable1, einVariable2;
            Console.Write("Geben Sie den Wert für Zahl 1 ein: ");
            einVariable1 = Convert.ToInt32 (Console.ReadLine());
            Console.Write("Geben Sie den Wert für Zahl 2 ein: ");
            einVariable2 = Convert.ToInt32(Console.ReadLine());
            //bitte jeweils in einer Zeile eingeben
            Console.WriteLine("Das Produkt der Zahlen ist {0}",
                Produkt(einVariable1, einVariable2));
        }
    }
}

```

4.5 Die Lösung könnte so aussehen:

```

/* #####
Lösung IV.5
##### */

using System;

namespace LoesIV5
{
    class Program
    {
        static int Produkt(int x, int y)
        {
            return (x * y);
        }

        static int Quadrat(int x)
        {
            return (Produkt(x,x));
        }
    }
}

```

```
static void Main(string[] args)
{
    int einVariable;
    Console.Write("Geben Sie eine Zahl ein: ");
    einVariable = Convert.ToInt32(Console.ReadLine());

    //bitte in einer Zeile eingeben
    Console.WriteLine("Das Quadrat der Zahl ist {0}",
        Quadrat(einVariable));
}
}
```


B. Glossar

Abbruchbedingung	Bei einer Schleife ist die Abbruchbedingung die Bedingung, die die Schleife steuert. Solange die Abbruchbedingung logisch „wahr“ ist, wird die Schleife ausgeführt.
Abfrage	Über eine Abfrage werden Daten aus einer Datenbank anhand bestimmter Kriterien selektiert.
Alternative Verzweigung	Bei einer alternativen Verzweigung werden sowohl Anweisungen vorgegeben, die ausgeführt werden sollen, wenn die Bedingung zutrifft, als auch Anweisungen, die ausgeführt werden sollen, wenn die Bedingung nicht zutrifft.
Anweisungsteil	Der Anweisungsteil ist der Teil einer Methode, der die Anweisungen enthält. Der Anweisungsteil wird immer durch geschweifte Klammern umfasst. Die Anweisungen im Anweisungsteil müssen durch ein Semikolon getrennt werden.
Argument	Ein Argument wird beim Aufruf einer Methode in den runden Klammern an die Methode übergeben. Statt des Begriffs Argument wird häufig auch der Begriff Parameter benutzt. Streng genommen bezeichnet der Begriff Parameter aber die Angaben im Kopf der Methode.
Arithmetik	Die Arithmetik ist ein Teilgebiet der Mathematik, das sich mit Zahlen und den für sie geltenden Rechenregeln befasst.
Ausnahme	Ausnahme ist die deutsche Übersetzung für <i>Exception</i> .
Ausnahmebehandlung	Durch die Ausnahmebehandlung können Sie in einem Programm auf Laufzeitfehler reagieren. Die Ausnahmebehandlung wird auch <i>Exception Handling</i> genannt.
Bedingung	Eine Bedingung steuert den weiteren Ablauf eines Programms – zum Beispiel bei einer Verzweigung oder bei einer Schleife.
Bezeichner	Ein Bezeichner ist der Name eines Datenobjekts. Für die Vergabe von Bezeichnern gibt es feste Regeln. So dürfen Sie zum Beispiel keine Schlüsselwörter verwenden und müssen einen Bezeichner mit einem Buchstaben oder dem Unterstrich _ beginnen.
Einfache Verzweigung	Bei einer einfachen Verzweigung werden Anweisungen nur dann ausgeführt, wenn eine Bedingung zutrifft.

Eingabeaufforderung	Die Eingabeaufforderung ist ein Teil von Windows, über den Konsolenprogramme gestartet werden können.
Endlosschleife	Eine Endlosschleife ist eine Schleife, die nie beendet wird. Endlosschleifen entstehen in der Regel durch Programmierfehler.
Exception	Eine <i>Exception</i> – eine Ausnahme – wird durch nicht definierte Zustände bei der Ausführung eines Programms ausgelöst. Dazu gehören zum Beispiel Divisionen durch Null bei ganzen Zahlen oder fehlgeschlagene Speicherreservierungen.
Exception Handler	Der <i>Exception Handler</i> besteht aus den Anweisungen, die beim Auftreten einer Ausnahme ausgeführt werden sollen.
Exception Handling	<i>Exception Handling</i> ist die englische Bezeichnung für Ausnahmebehandlung.
Flag	Als <i>Flag</i> wird bei der Programmierung eine Variable bezeichnet, die einen Zustand eindeutig anzeigt. Dafür werden häufig die Werte 0 und 1 beziehungsweise true und false benutzt. <i>Flag</i> bedeutet übersetzt „Flagge“.
Fußgesteuerte Schleife	Bei einer fußgesteuerten Schleife wird die Abbruchbedingung nach jedem Schleifendurchlauf geprüft. Eine fußgesteuerte Schleife wird daher mindestens einmal durchlaufen.
Geschachtelte Verzweigung	Bei einer geschachtelten Verzweigung werden mehrere Verzweigungen ineinander geschachtelt. Der if-Zweig einer Verzweigung enthält dann zum Beispiel selbst wieder eine Verzweigung.
Kontrollstruktur	Über eine Kontrollstruktur können Sie den Ablauf eines Programms steuern – zum Beispiel in Abhängigkeit von einer Bedingung bestimmte Anweisungen ausführen oder Anweisungen wiederholen.
Kopfgesteuerte Schleife	Bei einer kopfgesteuerten Schleife wird die Abbruchbedingung vor jedem Schleifendurchlauf geprüft.
Logische Verneinung	Siehe Logisches NICHT.
Logisches NICHT	Beim logischen NICHT wird ein Wert in sein Gegenteil verkehrt.
Logisches ODER	Beim logischen ODER werden zwei Ausdrücke miteinander verknüpft. Das Ergebnis ist wahr, wenn mindestens einer der beiden Ausdrücke wahr ist.

Logisches UND	Beim logischen UND werden zwei Ausdrücke miteinander verknüpft. Das Ergebnis ist nur dann wahr, wenn beide Ausdrücke wahr sind.
Main()	<code>Main()</code> ist die Hauptmethode eines C#-Programms.
Mehrfachauswahl	Bei der Mehrfachauswahl können zahlreiche Alternativen berücksichtigt werden. Die Mehrfachauswahl wird in C# durch die <code>switch ... case</code> -Konstruktion umgesetzt.
Methoden	Methoden beschreiben das Verhalten eines Objekts.
Modulo	Modulo liefert den Rest einer Division.
Nachlauf	Der Nachlauf ist Teil einer <code>for</code> -Schleife. Er wird nach jedem Schleifendurchlauf ausgeführt. Im Nachlauf wird in der Regel die Schleifenvariable verändert.
Objektorientierte Programmierung	Bei der objektorientierten Programmierung werden Daten- und Verhaltensaspekte gemeinsam betrachtet. Das wesentliche Element der objektorientierten Programmierung ist das Objekt.
Operand	Ein Operand ist der Teil eines Ausdrucks, auf den ein Operator wirkt.
Operator	Ein Operator ist ein Zeichen, das eine bestimmte Operation auslöst – zum Beispiel eine Rechenoperation. C# kennt verschiedene Arten von Operatoren – zum Beispiel <ul style="list-style-type: none"> • arithmetische Operatoren, • logische Operatoren und • Vergleichsoperatoren.
Parameter	Parameter geben an, welche Werte von einer Methode verarbeitet werden können. Sie werden im Kopf der Methode in runden Klammern angegeben. Der Begriff Parameter wird häufig auch für Argumente benutzt.
Quickinfos	Quickinfos sind kurze Hinweistexte. Sie werden normalerweise angezeigt, wenn der Mauszeiger einen kurzen Moment auf einem bestimmten Element stehen bleibt.
Reservierte Wörter	Reservierte Wörter sind Zeichenketten, die intern vom Compiler verwendet werden. Solche Zeichenketten dürfen Sie nicht selbst benutzen. Reservierte Wörter sind zum Beispiel <code>using</code> oder <code>while</code> .

Schleife	Über eine Schleife können Sie Anweisungen wiederholen lassen.
Schleifenfuß	Der Schleifenfuß ist das Ende einer Schleife.
Schleifenkörper	Der Schleifenkörper enthält die Anweisungen, die von einer Schleife ausgeführt werden sollen.
Schleifenkopf	Der Schleifenkopf ist der Anfang einer Schleife.
Syntaxfehler	<p>Syntaxfehler sind Verstöße gegen die Regeln der Programmiersprache – zum Beispiel ein fehlendes Semikolon oder eine falsche Klammer.</p> <p>Syntaxfehler werden vom Compiler bei der Übersetzung des Programms gemeldet.</p>
Testausdruck	Der Testausdruck ist Teil einer <code>for</code> -Schleife. Er steuert die Wiederholung der Schleife.
Tupel	Ein Tupel ist eine geordnete Menge von Werten.
Vorlauf	<p>Der Vorlauf ist Teil einer <code>for</code>-Schleife. Er wird einmal vor der Ausführung der Schleife ausgeführt.</p> <p>Im Vorlauf werden in der Regel Variablen initialisiert.</p>
Zählschleife	Zählschleife ist ein anderer Begriff für eine <code>for</code> -Schleife.

C. Literaturverzeichnis

Empfohlene Literatur

Kühnel, A. (2019). *C# 8 mit Visual Studio: Das umfassende Handbuch*. Spracheinführung, Objektorientierung, Programmier Techniken. 8. Aufl., Bonn: Rheinwerk.

Theis, T. (2019). *Einstieg in C# mit Visual Studio 2019*. Ideal für Programmieranfänger. 6. Aufl., Bonn: Rheinwerk.

D. Abbildungsverzeichnis

Abb. 2.1	Eine einfache if-Verzweigung	16
Abb. 2.2	Eine if ... else-Verzweigung	20
Abb. 2.3	Vergleich von if ... else (links) und if (rechts)	21
Abb. 2.4	Eine geschachtelte Verzweigung	23
Abb. 2.5	Geschachtelte Verzweigungen mit if ... else	24
Abb. 2.6	Die Entscheidungskette aus Code 2.4	26
Abb. 2.7	Die switch ... case-Konstruktion	27
Abb. 2.8	Darstellung der möglichen Fälle aus Code 2.6	32
Abb. 3.1	Die while-Schleife	37
Abb. 3.2	Die do ... while-Schleife	40
Abb. 3.3	Die for-Schleife	44
Abb. 4.1	Der Programmablauf für Code 4.2	62
Abb. 4.2	Ein ausgeblendeter Methodenkörper (links in der Mitte der Abbildung am Mauszeiger)	73
Abb. 4.3	Die Liste der Methoden (oben in der Mitte der Abbildung)	74

E. Tabellenverzeichnis

Tab. 1.1	Vergleichsoperatoren von C#	3
Tab. 1.2	Logische Operatoren von C#	5
Tab. 1.3	Wahrheitstabelle für das logische ODER	7
Tab. 1.4	Wahrheitstabelle für das logische UND	8
Tab. 2.1	Syntax der if-Verzweigung	19
Tab. 2.2	Syntax der if ... else-Verzweigung	22

F. Codeverzeichnis

Code 1.1	Beispiele für Vergleichsoperatoren	4
Code 1.2	Logisches NICHT	6
Code 1.3	Logisches ODER	8
Code 1.4	Logisches UND	9
Code 1.5	Kombination von logischen Operatoren mit Vergleichsoperatoren	10
Code 2.1	Einfache if-Verzweigung	16
Code 2.2	if-Verzweigung mit Anweisungsblock	18
Code 2.3	if ... else-Verzweigung	22
Code 2.4	Entscheidungsketten mit if ... else	25
Code 2.5	switch ... case-Konstruktion	29
Code 2.6	Verschachteltes switch ... case	31
Code 3.1	while-Schleife	38
Code 3.2	do ... while-Schleife	41
Code 3.3	Die umgebaute Schleife	42
Code 3.4	Noch eine umgebaute Schleife	43
Code 3.5	for-Schleife	45
Code 3.6	Von 0 bis 10 mit while	45
Code 3.7	Eine sehr kompakte for-Schleife	46
Code 3.8	break in einer Schleife	49
Code 3.9	Schleifenabbruch über eine Flag-Variable	51
Code 3.10	continue in einer Schleife	53
Code 3.11	if-Konstruktion statt continue	54
Code 3.12	Das Abfangen von Eingabefehlern	55
Code 4.1	Einfache Textausgabe	60
Code 4.2	Einfache Textausgabe mit Methoden	61
Code 4.3	Zwei Methoden mit Rückgabewert	64
Code 4.4	Undefinierte Rückgabe aus einer Methode (der Code lässt sich nicht übersetzen)	66
Code 4.5	Unterschiedliche Typen (der Code lässt sich ebenfalls nicht übersetzen)	66
Code 4.6	Das klappt nicht!	67
Code 4.7	Rückgabe mehrerer Werte über ein Tupel	68
Code 4.8	Eine Methode mit Argument	71
Code 4.9	Eine Methode mit zwei Argumenten	72

G. Medienverzeichnis

Video 4.1	Hilfen zum Arbeiten mit Methoden	74
------------------	--	----

H. Sachwortverzeichnis

A

Abbruchbedingung	36
Abfangen	
von ungültigen Eingaben	54
Anweisung	
break	47
continue	52
Anweisungsblock	18
Argument	61
Ausnahmebehandlung	54
Auswertungsblock	27

B

Bedingung	17
-----------------	----

C

case-Marke	27
case-Zweig	27

E

Endlosschleife	36
----------------------	----

F

Flag-Variable	50
---------------------	----

K

Klassenmethode	59
Kontrollstruktur	15

M

Mehrfachauswahl	
mit switch ... case	26
Methode	58
Main()	60
mit Argumenten	69
mit Rückgabewert	63
Methodenkörper	59

N

Nachlauf	44
----------------	----

O

ODER	
logisches	7
Operator	
logischer	5

P

Parameter	59
-----------------	----

R

Rückgabetyt	59
void	60

S

Schleife	36
fußgesteuerte mit do ... while	39
kopfgesteuerte mit while	37
Schleifenfuß	36
Schleifenkopf	36
Schleifenkörper	36
Schleifenvariable	38
Schlüsselwort	
return	63

T

Testausdruck	43
Tipp	
zum Arbeiten mit Methoden	73
Tupel	68

U

UND	
logisches	7, 8

V

Vergleichsoperator	3
==	3
Verneinung	
logische	5, 6

Verzweigung

geschachtelte	22
if-	15
if ... else-	20
Vorlauf	43

Z

Zählschleife

mit for	43
---------------	----

Zuweisungsoperator

=	3
---------	---

I. Einsendeaufgabe

Objektorientierte Software-Entwicklung mit C#

Code:

CSHP03D-XX1-N01

Name:	Vorname:
Postleitzahl und Ort:	Straße:
Studien- bzw. Vertrags-Nr.:	Lehrgangs-Nr.:

Fernlehrer/in:

Datum:

Note:

Unterschrift Fernlehrer/in:

Bitte reichen Sie Ihre Lösungen über die Online-Lernplattform ein oder schicken Sie uns diese per Post. Geben Sie bitte immer den Code zum Studienheft an (siehe oben rechts).

1. Schreiben Sie ein Programm, das eine Jahreszahl über die Tastatur abfragt und dann ausgibt, ob das Jahr ein Schaltjahr ist.

Die Überprüfung können Sie mit folgenden Regeln durchführen:

Ein Jahr ist kein Schaltjahr, wenn die Jahreszahl nicht durch 4 teilbar ist.

Ein Jahr ist ein Schaltjahr, wenn die Jahreszahl durch 4, aber nicht durch 100 teilbar ist.

Es ist ebenfalls ein Schaltjahr, wenn die Jahreszahl gleichzeitig durch 4, durch 100 und durch 400 teilbar ist.

Ein Beispiel:

Das Jahr 1964 war ein Schaltjahr. Die Jahreszahl lässt sich durch 4, aber nicht durch 100 teilen.

Das Jahr 1900 war kein Schaltjahr. Die Jahreszahl lässt sich zwar durch 4 und auch durch 100 teilen, aber nicht durch 400.

Sie können für die Überprüfung der Teilbarkeit den Modulo-Operator `%` und `if ... else`-Verzweigungen benutzen. Zur Erinnerung: Wenn eine Zahl `x` nicht glatt durch `y` teilbar ist, dann liefert der Ausdruck `(x % y)` einen Wert größer als 0.

Setzen Sie bei der Überprüfung der Teilbarkeit eine weitere Variable ein, die markiert, ob das Jahr ein Schaltjahr ist oder nicht. Werten Sie diese Variable am Ende des Programms aus und lassen Sie dann auf dem Bildschirm ausgeben, ob es sich um ein Schaltjahr handelt oder nicht.

Verwenden Sie bitte folgenden Programmkopf:

```
/* #####  
Einsendeaufgabe 3.1  
##### */
```

30 Pkt.

2. Schreiben Sie ein Programm, das von 1 bis 5 zählt und die Zahlen nebeneinander durch Kommas getrennt ausgibt. Vor der ersten Zahl und nach der letzten Zahl darf kein Komma stehen. Die Ausgabe soll durch eine `for`-Schleife erfolgen und so aussehen:

1,2,3,4,5

Verwenden Sie folgenden Programmkopf:

```
/* #####
Einsendeaufgabe 3.2
##### */
```

10 Pkt.

3. Erstellen Sie eine `while`-Schleife, die für die Zahlen 1 bis 25 jeweils das Doppelte des Wertes ausgibt. Für die Zahl 2 soll also der Wert 4 ausgegeben werden, für die Zahl 3 der Wert 6 und so weiter.

Verwenden Sie folgenden Programmkopf:

```
/* #####
Einsendeaufgabe 3.3
##### */
```

10 Pkt.

4. Programmieren Sie einen einfachen Taschenrechner. Er soll zwei Werte von der Tastatur einlesen und das Ergebnis einer Rechenoperation auf dem Bildschirm ausgeben. Dabei sollen auch Kommazahlen verarbeitet werden können. Als Rechenoperationen sollen Addition, Subtraktion, Division und Multiplikation möglich sein. Die Rechenoperationen sollen als eigene Methoden erstellt werden.

Verwenden Sie folgenden Programmkopf:

```
/* #####
Einsendeaufgabe 3.4
##### */
```

30 Pkt.

5. Erweitern Sie den Taschenrechner aus der Aufgabe 4 so, dass er Potenzen berechnen kann. Der erste eingelesene Wert soll dabei die Basis bilden und der zweite eingelesene Wert den Exponenten. Wenn Sie die Zahlen 2 und 3 eingelesen haben, soll der Taschenrechner also 2^3 rechnen.

Erstellen Sie für das Berechnen der Potenz eine eigene Methode. Die Potenz soll durch eine Schleife errechnet werden.

Verwenden Sie folgenden Programmkopf:

```
/* #####
Einsendeaufgabe 3.5
##### */
```

20 Pkt.

Gesamt: 100 Pkt.