

# Objektorientiertes Programmieren

Lerneinheit 2: Datentypen und Kontrollstrukturen

Autor: Bernd J. Krämer

Überarbeitung und Ergänzung: Silvia Schreier, Helga Huppertz

Abbildungen und Animationen: Volker Winkler, Silvia Schreier



# Inhaltsverzeichnis

## Kurseinheit 1

<b>1</b>	<b>Von der Aufgabenstellung zum Programm</b> .....	<b>1</b>
1.1	Motivation .....	1
1.2	Softwareentwicklung .....	2
1.3	EXKURS: Unified Modeling Language (UML) .....	4
<b>2</b>	<b>Anforderungsanalyse</b> .....	<b>9</b>
2.1	Fallstudie .....	9
2.2	Analyse der Anwendungswelt .....	10
2.3	Anwendungsfälle und Akteure .....	10
2.4	Beziehung zwischen Akteuren .....	13
2.5	Beziehung zwischen Anwendungsfällen .....	14
2.6	Anwendungsfallbeschreibungen .....	16
2.7	Datenlexikon .....	21
2.8	Pflichtenheft .....	22
<b>3</b>	<b>Einführung in die Objektorientierung</b> .....	<b>23</b>
3.1	Objekte und Klassen .....	23
3.2	Beziehungen .....	25
3.3	Kommunikation .....	28
3.4	Objektorientierte Analyse und Entwurf .....	29
3.5	UML-Klassendiagramm .....	31
3.6	UML-Objektdiagramm .....	33
<b>4</b>	<b>Einführung in die Algorithmik</b> .....	<b>35</b>
4.1	Algorithmen .....	35
4.2	Verhaltensbeschreibung und Kontrollstrukturen .....	40
<b>5</b>	<b>Programmiersprachen</b> .....	<b>44</b>
5.1	EXKURS: Entwicklungsgeschichte von Programmiersprachen ...	44
5.2	Eingabe, Verarbeitung, Ausgabe .....	46
5.3	Interaktive Programme .....	47
<b>6</b>	<b>Einführung in die Java-Programmierung</b> .....	<b>49</b>
6.1	Entwicklung und Eigenschaften der Programmiersprache Java ....	49
6.2	Erstellen, Übersetzen und Ausführen von Java-Programmen .....	50
<b>7</b>	<b>Zusammenfassung</b> .....	<b>54</b>
	<b>Lösungshinweise</b> .....	<b>57</b>
	<b>Index</b> .....	<b>69</b>

## Kurseinheit 2

---

<b>8</b>	<b>Praktischer Einstieg in die Java-Programmierung</b> .....	<b>71</b>
8.1	Ein erstes Programm .....	71
8.2	Klassen für die Praxis .....	73
8.3	Programmier- und Formatierhinweise .....	75
<b>9</b>	<b>Primitive Datentypen und Ausdrücke</b> .....	<b>76</b>
9.1	Ganze Zahlen .....	76
9.2	Gleitkommazahlen .....	79
9.3	Operatoren und Ausdrücke .....	81
9.4	Auswertung von Ausdrücken .....	83
9.5	Datentyp boolean .....	86
9.6	Datentyp char .....	87
<b>10</b>	<b>Variablen und Zuweisungen</b> .....	<b>89</b>
10.1	Variablen .....	89
10.2	Zuweisung .....	92
<b>11</b>	<b>Typanpassung</b> .....	<b>98</b>
11.1	Implizite Typanpassung .....	98
11.2	Explizite Typanpassung .....	103
<b>12</b>	<b>Anweisungen</b> .....	<b>108</b>
12.1	Blöcke .....	108
12.2	Kontrollstrukturen .....	110
<b>13</b>	<b>Bedingungsanweisungen</b> .....	<b>111</b>
13.1	Die einfache Fallunterscheidung .....	111
13.2	Die Mehrfach-Fallunterscheidung .....	116
<b>14</b>	<b>Wiederholungs- und Sprunganweisungen</b> .....	<b>119</b>
14.1	Bedingte Schleifen: while und do ... while .....	119
14.2	Die Zähl- oder for-Schleife .....	122
14.3	Strukturierte Sprunganweisungen: break und continue .....	126
<b>15</b>	<b>Gültigkeitsbereich und Sichtbarkeit von lokalen Variablen</b> .....	<b>131</b>
<b>16</b>	<b>Zusammenfassung</b> .....	<b>134</b>
	<b>Lösungshinweise</b> .....	<b>137</b>
	<b>Index</b> .....	<b>153</b>

---

# Abbildungsverzeichnis

Abb. 12.1-1	Struktur und Semantik eines Blocks. ....	109
Abb. 13.1-1	Struktur und Auswertung einer if-Anweisung.....	112
Abb. 13.1-2	if-Anweisung ohne Alternativzweig .....	112
Abb. 13.2-1	Mehrfach-Fallunterscheidung .....	117
Abb. 14.1-1	Semantik einer while-Schleife .....	120
Abb. 14.1-3	Semantik einer do-Schleife .....	122
Abb. 14.2-1	Semantik einer for-Schleife .....	124
Abb. 14.3-1	break-Anweisung in einer do-Schleife.....	127
Abb. 14.3-2	continue-Anweisung in einer for-Schleife .....	129
Abb. 15-1	Sichtbarkeit von Namen .....	133
Abb. 13.1-3	Verschachtelte if-Anweisung.....	144
Abb. 14.1-2	Aktivitätsdiagramm zu Bsp. while-Schleife .....	146
Abb. 14.1-4	Aktivitätsdiagramm zu Bsp. while-Schleife mit einer do-Schleife .....	148
Abb. 14.3-3	Aktivitätsdiagramm zu Bsp. break-continue .....	150
Abb. 14.3-4	Schleife mit break- und continue-Anweisung .....	151

## Tabellenverzeichnis

Tab. 9.1-1	Ganze Zahlen .....	76
Tab. 9.1-2	Werte von Literalen .....	78
Tab. 9.3-1	Grundlegende zweistellige arithmetische Operatoren .....	81
Tab. 9.4-1	Rangfolge von Operatoren .....	83
Tab. 9.4-2	Vergleichsoperatoren .....	85
Tab. 9.5-1	Boolesche Operatoren .....	87
Tab. 10.1-1	Schlüsselwörter .....	90
Tab. 11.1-1	Erweiterungsanpassungen .....	99

---

## Einführung

Computerprogramme verarbeiten Daten, die von Nutzerinnen oder technischen Einheiten wie Sensoren, Datenspeichern oder Kommunikationssystemen bereitgestellt werden. Aus diesen Eingabedaten erzeugen Programme neue Daten, die bedeutungsvolle Informationen für Menschen beinhalten und über geeignete Ausgabegeräte wie Bildschirm, Drucker oder Lautsprecher dargestellt werden. Programme können auch Daten erzeugen, die als Eingabe für andere Programme gebraucht werden, oder sie generieren Daten und Signale zur Steuerung technischer Geräte und Anlagen.

In dieser Kurseinheit beginnen wir mit der Einführung in die Programmierung mit Java. Dazu lernen Sie zunächst den grundlegenden Aufbau einfacher Java-Programme kennen, um von Anfang an die Beispiele und eigene Programme ausführen zu können.

Anschließend befassen wir uns in dieser Kurseinheit damit, wie **Daten** in Java-Programmen gespeichert und mit einfachen **Ausdrücken** verarbeitet werden können.

Neben dem zentralen Konzept der **Variablen** führen wir die Umsetzung der grundlegenden **Kontrollstrukturen**, die wir schon in der ersten Kurseinheit in Form von Aktivitätsdiagrammen kennen gelernt haben, in Java ein.

## Sprachspezifikation

Wie in der ersten Kurseinheit angekündigt, werden wir bei der Einführung von Elementen der Sprache Java angeben, wo diese in der Sprachspezifikation<sup>1</sup> definiert sind. Diese Angaben besitzen immer die folgende Form:

[JLS: § 5.6]

Diese Angabe bedeutet, dass Sie in der Sprachspezifikation im Abschnitt 5.6 die zugehörige Definition finden. Sie können diese Hinweise nutzen, um bei Detailfragen nachzulesen. Dies ist jedoch keine Voraussetzung für die erfolgreiche Bearbeitung des Kurses.

---

1 <http://java.sun.com/docs/books/jls/>

## Lernziele

- Die eingebauten Datentypen von Java verstehen und benutzen können.
- Die typische Verwendung von Datentypen beherrschen und die Bedeutung der Operatoren auf den Werten eines Datentyps verstehen.
- Korrekt formulierte Ausdrücke bilden und auswerten können.
- Wissen, wie man Variablen einführt und mit Werten belegt.
- Typanpassungen verstehen und verwenden können.
- Anweisungen und einfache Anweisungsfolgen bilden und auswerten können.
- Fallunterscheidungen und Schleifen programmieren können.
- Sichtbarkeitsbereiche von Namen verstehen.



# 8 Praktischer Einstieg in die Java-Programmierung

Das Erlernen einer Programmiersprache erfordert viel praktische Übung. Damit Sie von Anfang an das Gelernte praktisch nachvollziehen können, stellen wir in diesem Kapitel ein minimales Programmgerüst vor. Im weiteren Verlauf des Kurses können Sie dieses Gerüst jederzeit verwenden, um Beispiele nachzuvollziehen und Aufgaben zu lösen.

## 8.1 Ein erstes Programm

Rekapitulieren wir kurz, was wir aus der vorhergehenden Kurseinheit bereits über Java-Programme wissen:

- Der Quelltext von Java-Programmen befindet sich in Dateien mit der Endung `java`, wobei Dateinamen und die Namen der enthaltenen Klassen übereinstimmen.
- Nach der Übersetzung des Quelltexts kann die Programmausführung in der virtuellen Maschine gestartet werden.
- Dazu wird der virtuellen Maschine der Name einer Klasse mitgeteilt, die eine `main`-Methode enthält.
- Die Ausführung eines Java-Programms beginnt immer bei einer `main`-Methode.

Erstellen Sie mit einem Texteditor oder in der Entwicklungsumgebung Ihrer Wahl eine Datei mit dem Namen `ErsteKlasse.java` und dem folgenden Inhalt:

```
public class ErsteKlasse {  
  
    public static void main(String[] args) {  
        System.out.println("Herzlich willkommen!");  
    }  
}
```

Achten Sie darauf, dass die Datei nur diese Zeilen enthält. Falls durch Ihre Entwicklungsumgebung beim Anlegen der Datei bereits Quelltext-Teile automatisch entstanden sind, so löschen Sie diese zunächst. Zusätzliche Leerzeilen haben keine Auswirkungen.

Übersetzen Sie die Klasse `ErsteKlasse` und führen Sie sie aus. Sie erhalten die folgende Bildschirmausgabe:

```
Herzlich willkommen!
```

Die Bedeutung der einzelnen syntaktischen Bausteine, aus denen unser erste Programm besteht, werden wir im weiteren Verlauf des Kurses nach und nach kennen lernen. Für den Anfang genügt uns eine relativ grobe Betrachtung dieser kleinen Programmstruktur. Bevor wir diese in Angriff nehmen, ergänzen Sie das Programm noch wie folgt:

```
public class ErsteKlasse {                                // 1
    public static void main(String[] args) {            // 2
        // Gibt eine Begrüßung am Bildschirm aus
        System.out.println("Herzlich willkommen!");    // 3
    }                                                    // 4
}                                                        // 5
```

Speichern Sie die Änderungen ab, übersetzen Sie die Klasse `ErsteKlasse` erneut und führen Sie sie aus. Sie werden bemerken, dass sich das Verhalten des Programms nicht verändert hat.

**Kommentar** Bei den neu hinzugefügten Teilen handelt es sich um Kommentare, die vom Java-Übersetzer nicht verarbeitet werden. Sobald der Übersetzer in einer Zeile auf die Zeichengruppe `//` trifft, ignoriert er den Rest der Zeile. Kommentare werden eingesetzt, um die Bedeutung von Quelltext-Teilen zu dokumentieren. Im obigen Beispiel haben wir sie außerdem verwendet, um einzelne Zeilen für die folgende Diskussion mit Nummern zu versehen.

Strukturell besteht unser erstes Programm aus drei Komponenten:

- einer Klassendefinition,
- einer `main`-Methode und
- einer Anweisung.

**Klassendefinition** Die Klassendefinition beginnt in Zeile 1, die unter anderem den Namen der Klasse enthält. Die Zeile endet mit einer öffnenden geschweiften Klammer. Ähnlich wie Klammern in der Mathematik treten Klammern in Java-Programmen immer paarweise auf. Die zugehörige schließende geschweifte Klammer befindet sich in Zeile 5 und schließt die Klassendefinition ab. Das gesamte Programm besteht somit aus dieser Klassendefinition.

**main-Methode** Innerhalb der Klassendefinition ist die `main`-Methode definiert. Sie beginnt in Zeile 2, an deren Ende sich wiederum eine öffnende geschweifte Klammer befindet. In Zeile 4 endet die Definition der `main`-Methode mit einer schließenden geschweiften Klammer.

**Anweisung** In Zeile 3 enthält die `main`-Methode die Anweisung, eine Zeichenkette mit dem Inhalt `Herzlich willkommen!` am Bildschirm auszugeben. Diese Anweisung ist am Zeilenende durch ein Semikolon abgeschlossen.

## 8.2 Klassen für die Praxis

Die Klasse `ErsteKlasse` enthält ein minimales Programmgerüst. Jedes ausführbare Java-Programm besteht aus mindestens einer Klassendefinition, die eine `main`-Methode enthält. Eine `main`-Methode enthält üblicherweise eine oder mehrere Anweisungen, die nach dem Programmstart ausgeführt werden. In abgewandelter Form lässt sich ein solches Gerüst verwenden, um sämtliche in den folgenden Kapiteln vorgestellten Konzepte der Programmiersprache Java praktisch zu erproben.

Erstellen Sie eine Datei mit dem Namen `Testklasse.java` und dem folgenden Inhalt:

```
public class Testklasse { // 1
    public static void main(String[] args) { // 2
        // Breite und Laenge eines Rechtecks
        int breite = 1200; // 3
        int laenge = 27; // 4
        // Berechnung der Flaeche
        int flaeche = breite * laenge; // 5
        // Ausgabe der Flaeche
        System.out.println(flaeche); // 6
    } // 7
} // 8
```

Wie die Klasse `ErsteKlasse` besteht die Klasse `Testklasse` aus einer Klassendefinition (Zeilen 1 bis 8), die eine `main`-Methode enthält (Zeilen 2 bis 7). Die `main`-Methode der Klasse `Testklasse` enthält vier Anweisungen und drei Kommentare.

Die Bedeutung der Anweisungen wird sich Ihnen an dieser Stelle grob, nach dem Durcharbeiten von Kapitel 9 und Kapitel 10 vollständig erschließen. Die letzte Anweisung in Zeile 6 ähnelt der einzigen Anweisung in der Klasse `ErsteKlasse`. Sie bewirkt die folgende Bildschirmausgabe:

```
32400
```

Betrachten Sie die Klasse `Testklasse` als Vorlage für eigene praktischen Programmierübungen. Legen Sie sich eigene Testklassen an, indem Sie die Syntax der Klassendefinition und der `main`-Methode Zeichen für Zeichen übernehmen. Den Klassennamen können Sie abwandeln, am besten passend zur jeweiligen Aufgabenstellung. Denken Sie daran, dass Klassenname und Dateiname übereinstimmen müssen.

Alle Konstrukte, die Sie in dieser Kurseinheit kennen lernen, können innerhalb der `main`-Methode einer Klassendefinition ausgeführt werden. So lassen sich die korrekte Schreibweise für verschiedene Datentypen und die Auswertung von Ausdrücken (Kapitel 9) mit Hilfe von Ausgabeanweisungen der Form

```
System.out.println(0x1A);  
System.out.println(12 + 4 * 3);
```

erproben. Variablendeklarationen und Zuweisungen (Kapitel 10) lassen sich, wie in der Klasse `Testklasse` dargestellt, als Anweisungen innerhalb einer `main`-Methode ausführen:

```
int breite = 1200;
```

Ausgabeanweisung Dasselbe gilt für Anweisungssequenzen und verschiedene Kontrollstrukturen (ab Kapitel 12). Bedenken Sie immer, dass Sie ohne Ausgabeanweisung keine Rückmeldung vom Programm erhalten. Textausgaben erhalten Sie wie in der Klasse `ErsteKlasse` gezeigt, indem Sie den gewünschten Text in Anführungszeichen setzen:

```
System.out.println("Erste Berechnung ausgeführt.");
```

Soll ein Wert, etwa das Ergebnis einer Berechnung, ausgegeben werden, so geschieht dies ohne Anführungszeichen.

Schreiben Sie statt `println` lediglich `print`, so wird nach der Ausgabe keine neue Zeile begonnen. So resultieren die beiden Anweisungen

```
System.out.print("Die Antwort lautet: ");  
System.out.println(6 * 7);
```

in der Bildschirmausgabe:

```
Die Antwort lautet: 42
```

## 8.3 Programmier- und Formatierhinweise

Das Erlernen der Programmiersprache Java erfordert das Erlernen vieler Regeln und Gesetzmäßigkeiten, deren Einhaltung der Java-Übersetzer oder die virtuelle Maschine Ihnen zwingend vorschreibt.

Wir werden Ihnen darüber hinaus Programmier- und Formatierhinweise geben, deren Einhaltung nicht vom Übersetzer oder von der virtuellen Maschine gefordert wird. Sie sind angelehnt an die *Java Code Conventions*<sup>2</sup> und dienen dazu,

Programmier- und  
Formatierhinweise

- die Lesbarkeit von Programmen zu erhöhen,
- die Fehleranfälligkeit beim Programmieren zu senken und
- Konventionen zu vermitteln, die den Austausch mit anderen Programmierern erleichtern.

Programmier- und Formatierhinweise sind durch ein einheitliches Symbol gekennzeichnet, das unser erster wichtiger Hinweis demonstriert:

 Kommentieren Sie Ihr Programm von Anbeginn an!

## 9 Primitive Datentypen und Ausdrücke

Die Daten, die im Rechner verarbeitet werden können, werden in Programmiersprachen in Datentypen eingeteilt. Die Grundidee der Typisierung besteht darin, Werten im Speicher eines Rechners eine Bedeutung zuzuweisen.

### Definition 9-1: Datentyp

Datentyp *Ein Datentyp (engl. datatype) ist eine Menge von Werten und eine Menge von Operatoren, um diese Werte zu verknüpfen, zu vergleichen oder ineinander zu überführen.*

□

Schon im Mathematikunterricht in der Schule lernten wir, zwischen verschiedenen Arten von Zahlen, wie ganzen, reellen, rationalen und komplexen Zahlen, zu unterscheiden. Die Unterscheidung zwischen diesen Zahlenarten gründet auf dem Wertebereich, den sie repräsentieren, ihren Eigenschaften und den Operationen, die auf sie anwendbar sind. In ähnlicher Weise gehen wir mit Zahlen im Computer um.

In den folgenden Abschnitten führen wir die primitiven Datentypen von Java ein. Primitive Datentypen werden auch elementare oder einfache Datentypen genannt.

primitiver Datentyp  
elementarer Datentyp  
einfacher Datentyp

Datentypen verbinden ein Schlüsselwort wie `int` oder `char` mit einer Menge von Werten und Operatoren, die auf Werte des jeweiligen Typs anwendbar sind. Java unterstützt einfache Datentypen für Zahlen, boolesche Werte und Zeichen.

Neben den einfachen Datentypen gibt es in Java noch die Objekttypen. Einfache Datentypen sind in Java von der Sprache vordefiniert, Objekttypen werden durch Klassen- oder Schnittstellenvereinbarungen, auf die wir später eingehen werden, eingeführt.

### 9.1 Ganze Zahlen

In der Mathematik sind die Wertebereiche von Zahlen unendlich. Der Speicherplatz von Rechnern dagegen ist immer endlich. Daraus folgt, dass die Zahlenbereiche, die von Computern dargestellt werden können, ebenfalls endlich sind.

Ganze Zahlen Ganze Zahlen (engl. integer) werden bei Java in folgende vier Datentypen unterteilt [JLS: § 4.2.1]:

- |                             |                      |
|-----------------------------|----------------------|
| Datentyp <code>byte</code>  | • <code>byte</code>  |
| Datentyp <code>short</code> | • <code>short</code> |
| Datentyp <code>int</code>   | • <code>int</code>   |
| Datentyp <code>long</code>  | • <code>long</code>  |

Wertebereiche Die Wertebereiche dieser Datentypen und die Schreibweise für ihre Werte sind in Tab. 9.1-1 angegeben.

**Tab. 9.1-1:** Ganze Zahlen

Datentyp	Wertebereich	Schreibweise
byte	$-2^7, \dots, 0, \dots, 2^7-1$	-128, ..., 127
short	$-2^{15}, \dots, 0, \dots, 2^{15}-1$	-32768, ..., 32767
int	$-2^{31}, \dots, 0, \dots, 2^{31}-1$	-2147483648, ..., 2147483647
long	$-2^{63}, \dots, 0, \dots, 2^{63}-1$	-9223372036854775808L, ..., 9223372036854775807L

Die allgemeine Formel zur Berechnung der Wertebereiche ganzer Zahlen lautet:

$$-2^{8n-1}, \dots, 0, \dots, 2^{8n-1}-1$$

mit  $n=1$  für den Typ `byte`,  $n=2$  für den Typ `short`,  $n=4$  für den Typ `int` und  $n=8$  für den Typ `long`. [JLS: § 4.2.1]

Wir bezeichnen eine solche Folge von Ziffern als Ganzzahl-Literale (engl. integer literal), weil die zur Darstellung verwendeten Zeichen wörtlich genommen werden.

#### Definition 9.1-1: Literale

*Literale (engl. literal) sind Folgen von Zeichen, die Werte symbolisieren.* [JLS: § 3.10] Literal

□

Literale stellen Konstanten dar, da ihr Wert festgelegt ist und sich während der Ausführung eines Programms nicht verändert.

Dezimalliterale des Typs `int` sind die Ziffer 0 und jede Folge von Ziffern, die nicht mit 0 beginnt. int-Literal  
Dezimalliteral

In Java werden ganze Zahlen nicht nur als Dezimalzahlen, bestehend aus den Ziffern 0, . . . , 9, sondern auch als Folgen von Oktalzahlen oder Hexadezimalzahlen aufgeschrieben:

- Oktalliterale des Typs `int` werden aus der Ziffer 0 und einer nichtleeren Sequenz von Ziffern aus der Menge 0, 1, . . . , 7 gebildet. Oktalliteral
- Hexadezimalliterale des Typs `int` werden aus der Zeichenkombination 0X oder 0x und einer nichtleeren Sequenz von Zeichen aus der Menge der Ziffern 0, 1, . . . , 9 und der Buchstaben A, B, . . . , F oder a, b, . . . , f gebildet. Hexadezimalliteral

**Bemerkung 9.1-1: Interpretation von Literalen**

Diese Kennzeichnungen wurden bei Java eingeführt, um eine eindeutige Interpretation von Literalen zu gewährleisten. Anderenfalls könnte z. B. das Literal 123 nicht nur für den Dezimalwert 123, sondern auch für den Dezimalwert 83 stehen, wenn 123 als Oktalzahl aufgefasst wird, oder den Dezimalwert 291 darstellen, wenn 123 als Hexadezimalzahl aufgefasst wird.

□

Die Bedeutung einer ganzen Zahl ist der Wert, den das Literal bezeichnet, berechnet auf der Basis des zugehörigen Zahlensystem (oktal, dezimal oder hexadezimal).

**Beispiel 9.1-1: Werte von Literalen**

Tab. 9.1-2 stellt jeweils denselben Wert als dezimale, oktale bzw. eine hexadezimale ganze Zahl dar.

□

**Tab. 9.1-2:** Werte von Literalen

Literale	Berechnung	Dezimalwert
2476	$2 \cdot 10^3 + 4 \cdot 10^2 + 7 \cdot 10^1 + 6 \cdot 10^0$	2476
04654	$4 \cdot 8^3 + 6 \cdot 8^2 + 5 \cdot 8^1 + 4 \cdot 8^0$	2476
0x9AC	$9 \cdot 16^2 + 10 \cdot 16^1 + 12 \cdot 16^0$	2476

`long`-Literal    Literale des Typs `long` werden wie Literale des Typs `int` gebildet und mit dem Buchstaben `L` oder `l`, wie z. B. `67L`, beendet.

☞ Aus Gründen der Lesbarkeit wird empfohlen, bei `long`-Literalen den Großbuchstaben `L` statt klein `l` zu verwenden.

Literale, deren Betrag außerhalb des Wertebereichs von `int`-Literalen liegt, werden vom Übersetzer nur in Form von `long`-Literalen akzeptiert.

**Selbsttestaufgabe 9.1-1:**

Ist die folgende Aussage wahr oder falsch? Begründen Sie Ihre Antwort.

Das Literal `038` bezeichnet einen oktalen Wert vom Typ `int`.



**Selbsttestaufgabe 9.1-2:**

Welche der folgenden Zeichen sind gültige Literale vom Typ `byte`?

23L  
7  
128  
.4  
07

**Selbsttestaufgabe 9.1-3:**

Was ist der Dezimalwert von `0x2b9`?

- 254
- 697
- 725

**Selbsttestaufgabe 9.1-4:**

Warum erzeugt der Übersetzer bei dem folgenden Literal einen Übersetzungsfehler:

3456789012

Ganze Zahlen reichen nicht aus, um mit vielen praktischen Berechnungsproblemen, bei denen rationale Zahlen auftreten, umzugehen. Ein weiteres Paar primitiver Zahlentypen von Java, die Gleitkommazahlen, behebt diese Einschränkung.

## 9.2 Gleitkommazahlen

Java unterstützt zwei Datentypen für reelle Zahlen, wie wir sie aus der Mathematik kennen [JLS: § 4.2.3]:

- Gleitkommazahlen mit einfacher Genauigkeit vom Typ `float`
- Gleitkommazahlen mit doppelter Genauigkeit vom Typ `double`

Gleitkommazahlen  
Datentyp `float`  
Datentyp `double`  
Literele für  
Gleitkommazahlen

Literal für Gleitkommazahlen (engl. floating-point number) werden durch eine Sequenz folgender Zeichen dargestellt:

- einer optionalen Folge von Dezimalziffern (0, 1, ..., 9),
- einem Punkt „.“,
- einer optionalen Folge von Dezimalziffern (0, 1, ..., 9),

- einem optionalen Exponentenpart, der den Exponenten auf der Basis 10 darstellt, bestehend aus dem Buchstaben `e` oder `E` sowie einer Folge aus einem optionalen Vorzeichen („+“ oder „-“) und einer Dezimalzahl,
- einem optionalen Buchstaben `f` (`F`) oder `d` (`D`) zur Unterscheidung einer `float`- von einer `double`-Gleitkommazahl doppelter Präzision.

Sie unterliegen den folgenden Gestaltungsregeln:

- Vor oder nach dem Dezimalpunkt muss eine Ziffernfolge vorhanden sein,
- Entweder der Punkt, der Exponentenpart oder das `f` bzw. `D` am Ende müssen enthalten sein, damit das Literal als Gleitkommawert und nicht als ganze Zahl gewertet wird. [JLS: § 3.10.2]

Die hexadezimale Darstellung für Gleitkommazahlen werden wir nicht genauer betrachten.

Gleitkommazahlen sind vom Typ `double`, falls kein Buchstabe (`d`, `D`, `f`, `F`) am Ende des Literals angegeben ist.

Das heißt, dem Literal `24.456` wird vom Java-Übersetzer der Datentyp `double` zugeordnet.

#### **Bemerkung 9.2-1: Präzision**

*Die Präzision von Gleitkommazahlen ist beschränkt. Bei `float`-Zahlen beträgt die Genauigkeit ungefähr sieben Dezimalstellen, bei `double`-Zahlen ungefähr 15 Dezimalstellen. Operationen auf Gleitkommazahlen können die Genauigkeit weiter reduzieren. Betrachten Sie daher Gleitkommazahlen stets als Näherungen.*

□

#### **Beispiel 9.2-1: Rundungsfehler**

*Das Ergebnis der Division*

```
10000000000.0 / 3.0
```

*ist `3.3333333333333335E9`. Das Ergebnis der Multiplikation*

```
100.0 * 4.35
```

*ist `434.99999999999994`.*

□

#### **Selbsttestaufgabe 9.2-1:**

*Welche der folgenden Zeichen sind gültige Literale vom Typ `float`?*

```
24.E-13
```

```
.4e0
```

```
1.341f
```

```
.E-45
```

**Selbsttestaufgabe 9.2-2:**

Bestimmen Sie den Typ der folgenden numerischen Literale:

- 1234
- 2f
- 44.45
- 2e1f
- 0x2e1f
- .3
- 2124L
- 41D
- 6d
- 0x88
- 3E10

Einfach nur Werte niederzuschreiben, führt nicht weit. Wir benötigen darüber hinaus Konzepte, um die Funktionen über solche Werte darzustellen. Operatoren in Java, die ähnlich aussehen wie diejenigen, die uns bereits aus dem Mathematikunterricht bekannt sind, erlauben es uns, solche Funktionen über Werte zu formulieren.

## 9.3 Operatoren und Ausdrücke

In Java können wir arithmetische Ausdrücke aus Folgen von Literalen, Operatoren und runden Klammern „(“ und „)“ bilden.

### Definition 9.3-1: Operator

*Ein Operator führt eine Funktion (wie Negation, Addition, Subtraktion, Multiplikation usw.) auf seinem oder seinen Operanden aus und liefert einen Wert zurück. Der Typ des Ergebniswerts wird durch den oder die Typen der Operanden bestimmt. Der Ergebniswert hängt vom Operator und von den Werten seiner Operanden ab.*

Operator

*Die einstelligen Operatoren + und – bestimmen den positiven bzw. negativen Wert eines arithmetischen Ausdrucks. Die zweistelligen arithmetischen Operatoren:*

arithmetischer Operator

+ , - , \* , / , %

*stellen die Addition, Subtraktion, Multiplikation, ganzzahlige Division bzw. Restwertdivision (engl. modulo division) für zweistellige Operatoren dar. Diese Operatoren und ihre Bedeutungen sind zusammen mit einem Beispiel in Tab. 9.3-1 angegeben. [JLS: § 3.12, § 4.2.2, § 4.2.4]*

□

**Tab. 9.3-1:** Grundlegende zweistellige arithmetische Operatoren

Verwendung	Bedeutung	Beispiel	Ergebnis
$op1 + op2$	$op1$ und $op2$ werden addiert	$10 + 3$ $9.5 + 3.8$	13 13.3
$op1 - op2$	$op2$ wird von $op1$ subtrahiert	$10 - 3$ $9.5 - 3.8$	7 5.7
$op1 * op2$	$op1$ und $op2$ werden miteinander multipliziert	$10 * 3$ $9.5 * 3.8$	30 36.1
$op1 / op2$	$op1$ wird durch $op2$ dividiert, bei Ganzzahlen wird der Rest ignoriert	$10 / 3$ $11 / 3$ $9.5 / 3.8$	3 3 2.5
$op1 \% op2$	ergibt den Rest der ganzzahligen Division von $op1$ durch $op2$	$10 \% 3$ $11 \% 3$	1 2

Diese Operatoren sind in Java auf alle Ganz- und Gleitkommazahltypen anwendbar, solange die beiden Operanden vom gleichen Typ oder von verträglichen Typen (vgl. Kapitel 11) sind.

Operatoren, die auf mehr als einen Typen anwendbar sind, nennen wir überladen (engl. overloaded).

#### **Bemerkung 9.3-1: Operanden**

Operand Neben Literalen werden wir im Laufe des Kurses noch weitere Alternativen für Operanden in Ausdrücken kennen lernen.

□

#### **Definition 9.3-2: Arithmetischer Ausdruck**

arithmetischer Ausdruck Ein arithmetischer Ausdruck (engl. arithmetic expression) wird gemäß folgender Regeln gebildet [JLS: § 15]:

- Jede Variable eines numerischen Typs und jedes Literal eines numerischen Typs ist ein arithmetischer Ausdruck.
- Wenn  $x$  ein arithmetischer Ausdruck ist, dann ist auch  $(x)$  ein arithmetischer Ausdruck.
- Wenn  $x$  ein arithmetischer Ausdruck und  $eop$  ein einstelliger arithmetischer Operator ist, der vor dem Operanden steht, dann ist auch  $eop x$  ein arithmetischer Ausdruck.
- Wenn  $x$  ein arithmetischer Ausdruck und  $eop$  ein einstelliger arithmetischer Operator ist, der nach dem Operanden steht, dann ist auch  $x eop$  ein arithmetischer Ausdruck.
- Wenn  $x$  und  $y$  arithmetische Ausdrücke sind, und  $zop$  ein zweistelliger arithmetischer Operator, dann ist auch  $x zop y$  ein arithmetischer Ausdruck.

□

☞ Der Java-Übersetzer akzeptiert Leerzeichen zwischen Operanden, Operatoren und Klammern, verlangt sie aber nicht. Nach weit verbreiteter Konvention werden zwischen den einstelligen Operatoren +, - und Operanden sowie zwischen Klammern und Operanden keine Leerzeichen verwendet. Bei zweistelligen Operatoren werden die Operanden mit Leerzeichen abgetrennt.

### Selbsttestaufgabe 9.3-1:

Überprüfen Sie, ob der arithmetische Ausdruck

$$12 * 5 \% - 3 - 5 / 3 + 7$$

syntaktisch wohlgeformt ist, und versuchen Sie ihn auszuwerten, falls Ihre Antwort positiv ausfällt!

## 9.4 Auswertung von Ausdrücken

Wenn in einem Ausdruck mehrere Operatoren auftreten, wendet der Java-Übersetzer Präzedenzregeln an, um den Ausdruck gemäß der Rangfolge der auftretenden Operatoren auszuwerten, es sei denn, dass durch Klammern eine andere Reihenfolge vorgegeben wurde. Die Präzedenzregeln sind in Tab. 9.4-1 angegeben. (Die Tabelle enthält auch einige Operatoren, die wir erst in einem späteren Abschnitt besprechen werden.) Die Operatoren in der ersten Zeile haben den höchsten Rang, die in der letzten den niedrigsten. Operatoren gleichen Rangs werden von links nach rechts ausgewertet.

Präzedenzregeln

Rangfolge von Operatoren


**Tab. 9.4-1:** Rangfolge von Operatoren

Rangfolge (von oben nach unten)	Kategorie	Bedeutung
op++ op--	arithmetisch	Postinkrement Postdekrement
++op --op +op -op ~op !op (type)	arithmetisch/ boolesch	Präinkrement Prädekrement positives Vorzeichen negatives Vorzeichen bitweises Komplement logische Negation Typumwandlung
* / %	arithmetisch	Multiplikation Division Divisionsrest
+ -	arithmetisch	Addition Subtraktion

Fortsetzung auf der nächsten Seite

Fortsetzung von der vorhergehenden Seite		
<< >> >>>	shift	bitweises Verschieben
< > <= >=	relational	kleiner größer kleiner oder gleich größer oder gleich
== !=	Gleichwertigkeit	gleich ungleich
&	boolesch	Boolesches/Bitweises Und (vollständige Auswertung)
^	boolesch	boolesches/bitweises exclusives Oder
	boolesch	Boolesches/Bitweises Oder (vollständige Auswertung)
&&	boolesch	Logisches Und (unvollst. Auswertung)
	boolesch	Logisches Oder (unvollst. Auswertung)
? :	Bedingung	Bedingte Auswertung
= += -= /= %= ...	Zuweisung	Zuweisung Zuweisung nach Inkrement Zuweisung nach Dekrement Zuweisung nach Division Zuweisung nach Restbildung ...

In Ausdrücken können Klammern benutzt werden, um die Reihenfolge der Berechnung bei untergeordneten Ausdrücken festzulegen.

 Zusätzliche Klammern, die die Auswertungsreihenfolge nicht verändern, können zur Erhöhung der Lesbarkeit verwendet werden.

#### **Definition 9.4-1: Auswertung arithmetischer Ausdrücke**

Auswertungsreihenfolge

*Ein arithmetischer Ausdruck wird in einer strikten Reihenfolge gemäß folgender Regeln berechnet [JLS: § 15.7]:*

- *Ausdrücke, die in Klammern eingeschlossen sind, werden zuerst berechnet; im Falle von Verschachtelungen von innen nach außen. Klammersausdrücke erhalten somit die höchste Priorität bei der Auswertung.*
- *Zweistellige Operatoren werden von links nach rechts ausgewertet.*
- *Einstellige Operatoren werden von rechts nach links ausgewertet.*

- Anschließend werden die Operatoren ihrem Vorrang entsprechend (s. Tab. 9.4-1) ausgewertet.
- Wenn ein Ausdruck mehrere Operatoren des gleichen Vorranges enthält, werden die Operatoren von links nach rechts berechnet.

□

### Beispiel 9.4-1: Auswertung arithmetischer Ausdrücke

Unter Berücksichtigung der Regeln in Definition 9.4-1 wird der Ausdruck

$$17 * 2 \% - (4 - 11) / (-3 + 5)$$

in folgender Reihenfolge berechnet (gelesen von oben nach unten, der als Nächstes auszuwertende Unterausdruck ist unterstrichen):

Auswertungsreihenfolge	Regel	Auswertungsrichtung
$17 * 2 \% - \underline{(4 - 11)} / (-3 + 5)$	( )	von links nach rechts
-> $17 * 2 \% - (-7) / \underline{(-3 + 5)}$	( )	von links nach rechts
-> $17 * 2 \% - \underline{(-7)} / 2$	-	von rechts nach links
-> $\underline{17 * 2} \% 7 / 2$	*	von links nach rechts
-> $\underline{34 \% 7} / 2$	%	von links nach rechts
-> $\underline{6} / 2$	/	von links nach rechts
-> 3		

□

Auf Zahlen können außerdem Vergleichsoperatoren angewendet werden. In Java finden wir die Vergleichsoperatoren

Vergleichsoperator

==, !=, >, >=, <, <=,

um zwei Werte gleichen oder verträglichen Typs miteinander zu vergleichen. Syntax und Bedeutung dieser Operatoren sind in Tab. 9.4-2 angegeben. [JLS: § 15.20, § 15.21]

**Tab. 9.4-2:** Vergleichsoperatoren

Verwendung	Bedeutung
$op1 == op2$	$op1$ und $op2$ werden auf Gleichheit getestet
$op1 != op2$	$op1$ und $op2$ werden auf Ungleichheit getestet
$op1 > op2$	testet, ob $op1$ größer als $op2$ ist
$op1 >= op2$	testet, ob $op1$ größer als oder gleich $op2$ ist
$op1 < op2$	testet, ob $op1$ kleiner als $op2$ ist
$op1 <= op2$	testet, ob $op1$ kleiner als oder gleich $op2$ ist

Während das Ergebnis arithmetischer Ausdrücke ein numerischer Wert ist, resultieren Vergleichsoperationen in einem Wahrheitswert. Den zugehörigen Datentyp lernen wir im folgenden Abschnitt kennen.

**Selbsttestaufgabe 9.4-1:**

Werten Sie die folgenden Ausdrücke aus:

$$9 + 3 * 4$$

$$78 \% 4 + 2$$

$$(3 + 43 * 2) \% 7$$
**Selbsttestaufgabe 9.4-2:**

Welche der untenstehenden Ausdrücke sind mit dem Ausdruck:

$$12 * 5 \% - 3 - 5 / 3 + 7$$

äquivalent?

1.  $(( (12 * 5) \% (-3)) - (5 / 3)) + 7$
2.  $(12 * (5 \% (-3))) - (5 / 3) + 7$
3.  $((12 * (5 \% (-3 - 5))) / 3) + 7$
4.  $(12 * (5 \% (-3)) - (5 / 3)) + 7$

## 9.5 Datentyp boolean

**Datentyp `boolean`** Neben den numerischen Datentypen offeriert Java auch den primitiven Datentypen `boolean`, um mit Wahrheitswerten in logischen Ausdrücken umzugehen und um das Ergebnis von Vergleichsoperatoren auf numerischen und anderen Werten darzustellen.

### Definition 9.5-1: Datentyp `boolean`

**boolean-Literal** *Der Datentyp `boolean` umfasst die beiden Literale `true` und `false` sowie die folgenden booleschen Operatoren:*

- den einstelligen Präfixoperator `!`, der eine logische Negation beschreibt,
- fünf zweistellige Operatoren `&&`, `||`, `&`, `|` und `^`, um logische Ausdrücke aufzubauen und
- zwei zweistellige Vergleichsoperatoren `==` und `!=`, die die logische Gleichheit bzw. Ungleichheit beschreiben.

**boolescher Operator** *Alle Operatoren vom Typ `boolean` sind mit ihrer Bedeutung und Anwendungsbeispielen in Tab. 9.5-1 angegeben. [JLS: § 3.10.3, § 4.2.5, § 15.22.2, § 15.23, § 15.24]*

□



**Tab. 9.5-1:** Boolesche Operatoren

Verwendung	wird true, wenn
<code>op1 &amp;&amp; op2</code>	op1 und op2 beide true sind; op2 wird nur ausgewertet, wenn op1 zu true evaluiert
<code>op1    op2</code>	op1 oder op2 true ist; op2 wird nur ausgewertet, wenn op1 false ist
<code>op1 &amp; op2</code>	op1 und op2 beide true sind; op1 und op2 werden immer ausgewertet
<code>op1   op2</code>	op1 oder op2 true ist; op1 und op2 werden immer ausgewertet
<code>op1 ^ op2</code>	op1 und op2 zu verschiedenen Wahrheitswerten evaluieren
<code>!op</code>	op false ist

Bei Ausdrücken, die über die Operatoren `&&` und `||` aufgebaut sind, wird der rechtsseitige Operator nicht ausgewertet, wenn das Ergebnis des Ausdrucks durch die Auswertung des linksseitigen Operators bereits feststeht. Dies bedeutet, dass der Operand `d` im Ausdruck `e && d` nicht mehr ausgewertet wird, falls Operand `e` zum Ergebnis `false` evaluiert, weil nach den Gesetzen der Logik der Ausdruck „falsch UND `d`“ falsch ist, unabhängig vom Wahrheitswert der Variablen `d`. Entsprechend wird der rechte Operand im Ausdruck `e || d` nicht ausgewertet, falls der Operand `e` zu `true` evaluiert.

### Selbsttestaufgabe 9.5-1:

Werten Sie die folgenden Ausdrücke aus:

```

3 / 6 > 0
(13 % 4) == (17 % 4)
(5 * 7 == 35) && (2 + 3 == 10)
true && false || true
3 > 4 ^ 7 >= 3 + 4
true && (3 < 4 || 4 > 5)
false || 4 == 2 * 2 ^ 2 == 1 + 1

```

## 9.6 Datentyp char

Java bietet den einfachen Datentyp `char` an, der alle Unicode-Zeichen umfasst und seine Werte vorzeichenlos speichert. Unicode<sup>3</sup> ist ein internationales Codiersystem für Zeichen, das es ermöglicht, mehr als 65000 Zeichen darzustellen. Unicode-Zeichen umfassen:

Datentyp `char`  
Unicode

3 <http://www.unicode.org/charts/>

- Groß- und Kleinbuchstaben,
- Ziffern,
- Interpunktionszeichen,
- Sonderzeichen für Operatoren,
- Sonderzeichen und -buchstaben, die in verschiedenen Sprachen benutzt werden, und
- Steuerzeichen, die es ermöglichen, Aufteilung und Umbruch einer Computerausgabe zu bestimmen.

### Definition 9.6-1: Zeichen (`char`)

`char`-Literal  
Zeichen

Zeichen erscheinen zwischen einzelnen Anführungszeichen (') wie:

'a', 'B', '§', oder '- '.

Steuerzeichen beginnen mit dem Rückwärts-Schrägstrich (engl. *backslash*) „\“.  
[JLS: § 3.10.4]

□

### Beispiel 9.6-1: Steuerzeichen

Steuerzeichen

Die Zeichen '\n', '\t' und '\b' stehen zum Beispiel für die Steuerzeichen „neue Zeile“, „Tabulatorsprung“ bzw. „Rücktaste“.

□

Auch auf Werte des Typs `char` sind die sechs in Tab. 9.4-2 angegebenen Vergleichsoperatoren anwendbar. Sie ermöglichen es, zwei Zeichen auf ihre Gleichheit (==) oder Ungleichheit (!=) und ihre Reihenfolge im Unicode (mittels <, <=, >, >=) hin zu überprüfen.

### Selbsttestaufgabe 9.6-1:

Welche Ergebnisse liefern die folgenden Vergleiche?

```
'a' < 'Z'
'a' == 'A'
'&' <= '5'
'#' > '$'
'e' < 'u'
```

**Lösungshinweis:** Nutzen Sie die *Latin-Unicode-Tabelle*<sup>4</sup>. Die Nummern unter den Zeichen sind die Werte der Zeichen in hexadezimaler Schreibweise.

## 10 Variablen und Zuweisungen

Bisher haben wir Ausdrücke mit konkreten Zahlen oder Zeichen formuliert. Um ganze Problemklassen zu lösen, müssen wir diese ähnlich wie in der Mathematik variabel halten können. Um zum Beispiel die Mehrwertsteuer von 19% von 100 € zu berechnen, würden wir

$$100 * 0.19$$

schreiben. Will man jedoch für einen beliebigen Preis  $p$  die Mehrwertsteuer von 19% berechnen, so könnte man dies mit

$$p * 0.19$$

ausdrücken. Dieses Kapitel erläutert die Formulierung und Verwendung solcher allgemeinen Ausdrücke und stellt die dafür benötigten Sprachmittel vor.

### 10.1 Variablen

In mathematischen Formeln stellen Variablen Bezeichner zur Verfügung, die beliebige Werte annehmen können. Variablen werden in der Mathematik meist ohne ausdrückliche Vereinbarung (engl. declaration) benutzt, weil der Wertebereich, den sie repräsentieren, aus dem Zusammenhang, in dem sie benutzt werden, abgeleitet werden kann. In einem bestimmten Kontext repräsentiert jedoch jedes Auftreten einer mathematischen Variablen  $x$  den gleichen Wert.

Variablen spielen auch bei Programmiersprachen eine wichtige Rolle. Verschiedene Vorkommen der gleichen Variablen können in einem Programmkontext jedoch verschiedene Werte annehmen.

#### Definition 10.1-1: Variablen

*Eine Variable besteht aus einem Namen und einem Wert. [JLS: § 4.12]*

Variable

□

Variablen in Programmiersprachen agieren wie Behälter, die zu einem bestimmten Zeitpunkt jeweils bestimmte Werte bereit halten. Der Wert einer Variablen kann sich über die Lebensdauer eines Programms hin verändern. Der Wert einer Variablen kann mit Hilfe des Namens abgefragt und verändert werden. Die zulässige Wertemenge, aus der eine Variable einen Wert annehmen kann, wird durch ihren Datentyp bestimmt. Der Datentyp wird bei der Vereinbarung einer Variablen festgelegt. Anders als in der Mathematik müssen Variablen in Java-Programmen ausdrücklich vereinbart werden, bevor man sie benutzen kann.

**Definition 10.1-2: Variablenvereinbarung**

Variablenvereinbarung  
Variablendeklaration

*Eine Variablenvereinbarung oder Variablendeklaration ist eine Anweisung, die einen Namen mit einem Datentyp verknüpft. Die einfachste Form einer Variablendeklaration verbindet einen Namen `name` mit einem Datentyp `type`. Jede Deklaration wird durch ein Semikolon abgeschlossen. [JLS: § 14.4]*

```
type name;
```

□

In Java müssen Variablennamen nach bestimmten Vorgaben gebildet werden.

**Definition 10.1-3: Aufbau von Variablennamen**


Variablenname  
Schlüsselwörter

*Namen von Variablen bestehen aus einer Folge von Zeichen. Als Zeichen sind alle Unicodezeichen zugelassen. Das erste Zeichen darf jedoch keine Ziffer (0-9) sein. Ein Variablenname darf nicht mit einem reservierten Schlüsselwort der Sprache Java (s. Tab. 10.1-1) oder den Literalen `true`, `false` und `null` übereinstimmen. (Es gelten noch einige weitere Einschränkungen, die wir im Rahmen des Kurses nicht näher erläutern.) [JLS: § 3.8]*

□

**Tab. 10.1-1: Schlüsselwörter**

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

 Im Rahmen des Kurses verwenden wir für Variablennamen nur die Zeichen A-Z, a-z und Ziffern.

**Beispiel 10.1-1: Variablendeklaration**

*Der Ausdruck*

```
int menge;
```

*vereinbart eine Variable mit Namen `menge` vom Typ `int`, deren Wert noch unbestimmt ist.*

□

**Bemerkung 10.1-1: Eindeutigkeit von Variablen**

*In jedem speziellen Umfeld darf eine Variable nur einmal vereinbart werden.*

□

**Beispiel 10.1-2: Ungültige Variablenvereinbarung**

```
int breite;
double breite;
```

Die zweite Vereinbarung für den Namen *breite* ist unzulässig - unabhängig vom gewählten Typ.

□

Für eine Folge von Variablendeklarationen des gleichen Typs, wie

```
double einkaufspreis;
double verkaufspreis;
```

akzeptiert der Java-Übersetzer auch eine verkürzte Schreibweise

```
double einkaufspreis, verkaufspreis;
```

wobei aufeinander folgende Variablennamen durch Kommata voneinander getrennt sind.

☞ Aus Gründen der Übersichtlichkeit raten wir Ihnen jedoch, von dieser Verkürzung keinen Gebrauch zu machen.

☞ Nach den Java-Konventionen beginnen Variablennamen mit einem Kleinbuchstaben. Namen sollten i. d. R. beschreibend und selbsterklärend sein. Namen, die mehr als ein Wort umfassen, sollten mit gemischten Buchstaben geschrieben werden, wobei Großbuchstaben für Wortanfänge in zusammengesetzten Namen verwendet werden:

```
int anzahlBestellungen;
boolean istPremiumkunde;
```

Im Rahmen dieses Kurses verwenden wir für kurze Beispiele ohne Kontext gelegentlich Variablennamen aus einem einzelnen Zeichen. [JLS: § 6.8.7]

**Selbsttestaufgabe 10.1-1:**

Was ist bei den folgenden Deklarationen falsch und warum?

```
int x y z;
double currency Rate;
int null;
int x, int y;
bool b
int 3times;
```

**Selbsttestaufgabe 10.1-2:**

*Welche der folgenden Variablennamen sind ungültig und warum?*

```
myName, 3D, HEIGHT, this, this_position,
any_Float, thisMonth, 4you, x3
```

**10.2 Zuweisung**

Eine Variable, die in einem Ausdruck vorkommt, muss einen Wert haben. Der Wert einer Variablen in einem Programm kann durch eine Zuweisung bestimmt und verändert werden.

**Definition 10.2-1: Zuweisung**

Zuweisung *Eine Zuweisung ist ein Ausdruck und zugleich eine Anweisung der Form:*

```
varName = expression;
```

*Wenn  $w(expression)$  den Wert des Ausdrucks  $expression$  bezeichnet, dann gilt unmittelbar nach der Ausführung der Zuweisung:*

```
 $w(varName = expression) == w(varName) == w(expression)$ 
```

*[JLS: § 15.26, § 16]*

□

**Beispiel 10.2-1: Zuweisung**

*Sei  $x$  eine Variable vom Typ `int`. Die Zuweisung*

```
x = 42;
```

*hat zur Folge, dass der Variablen  $x$  der Wert 42 zugewiesen wird.*

*Die Zuweisung*

```
x = 158 - 99;
```

*hat zur Folge, dass der Variablen  $x$  der Wert 59 zugewiesen wird, der sich als Ergebnis der Auswertung des Ausdrucks `158 - 99` ergibt.*

*Die Zuweisung*

```
x = y;
```

*hat zur Folge, dass der Variablen  $x$  der Wert der Variablen  $y$  zugewiesen wird. Dies setzt voraus, dass die Variable  $y$  zuvor deklariert und bereits mit einem Wert belegt worden ist.*

□

Zuweisungsoperator In Java ist das Zuweisungssymbol „`=`“ ein Operator, und jede Zuweisung ist ein Ausdruck. Sein Wert ist der Wert des rechtsseitigen Operanden.

Eine Variable ist auch ein Ausdruck. Der Wert des Ausdrucks ist der Wert, der in der Variablen gespeichert ist.

Variablen können initialisiert, d. h. bei ihrer Deklaration mit Anfangswerten belegt werden, wenn sie mit einer Zuweisung verbunden werden. Eine Initialisierung hat die Form: Initialisierung

```
type varName = expression;
```

Der Datentyp des Ausdrucks `expression` muss dem vereinbarten Typ `type` entsprechen.

Ein Zuweisungsausdruck wird von rechts nach links berechnet. Deshalb ist es möglich, dass dieselbe Variable auf beiden Seiten des Zuweisungsoperators auftritt. Auf diese Weise kann einer Variablen in Abhängigkeit von ihrem bisherigen Wert ein neuer Wert zugewiesen werden.

### **Beispiel 10.2-2: Auswertung und Zuweisung für dieselbe Variable**

*Die Variable  $x$  sei vom Typ `int` und habe den Wert 3. Dann wird durch die Anweisung:*

```
x = 2 + 3 * x;
```

*der Wert von  $x$  auf 11 geändert, weil die Auswertung des Ausdrucks  $3 * x$  das Ergebnis 9 hat und  $2 + 9$  den Wert 11 ergibt.*

□

Java ermöglicht einige abgekürzte Schreibweisen für Zuweisungen der Form: abgekürzte Schreibweise

```
variable = variable operator (expression)
```

bei der beide Variablen auf der rechten und linken Seite des Zuweisungsoperators identisch sind. Sie können abgekürzt werden zu:

```
variable operator= expression
```

**Beispiel 10.2-3: Operator für verkürzte Schreibweisen von Zuweisungen***Seien*

```
int x = 2;
int y = 5;
```

*Der Ausdruck*

```
x += 3;
```

*ist die Abkürzung für den Ausdruck*

```
x = x + 3;
```

*Die Variable  $x$  nimmt als Ergebnis der Ausführung der Zuweisung den Wert 5 an.**Der Ausdruck*

```
y += 6 % 4;
```

*steht abkürzend für*

```
y = y + (6 % 4);
```

*Nach Ausführung der Zuweisung hat  $y$  den Wert 7.*

□

**Bemerkung 10.2-1: Auswertungsreihenfolge**

*Beachten Sie, dass Operatoren für die verkürzte Schreibweise von Zuweisungen zuletzt ausgewertet werden. Wir können deshalb die verkürzte Schreibweise nicht in jedem Fall anwenden, wo rechts und links des Zuweisungsoperators dieselbe Variable auftritt. So lässt sich die Zuweisung in Beispiel 10.2-2 nicht verkürzt schreiben, da ansonsten die Auswertungsreihenfolge verändert würde.*

□

In Java kann eine Variable als endgültig oder `final` erklärt werden. Der Wert einer finalen Variablen kann, nachdem ihr zum ersten Mal ein Wert zugewiesen wurde, nicht mehr verändert werden. Um eine Variable als finale Variable zu deklarieren, wird das Schlüsselwort `final` dem Variablentyp vorangestellt.

Der Wert einer finalen Variablen kann bei der Deklaration oder zu einem späteren Zeitpunkt der Programmausführung festgesetzt werden. Ein zweiter Versuch, einer finalen Variable einen Wert zuzuweisen, würde vom Übersetzer mit einer Fehlermeldung zurückgewiesen werden. [JLS: § 4.12.4]



**Beispiel 10.2-4: Finale Variable `final`**

```
final double PI = 3.147;
```

Die oben stehende Deklaration führt eine finale Variable ein und belegt sie mit dem Wert `3.147`. Die nachstehende Folge von Anweisungen zeigt eine Situation, in der eine finale Variable zunächst deklariert, aber noch nicht initialisiert wird und erst in einer nachfolgenden Anweisung mit ihrem endgültigen Wert belegt wird:

```
final int BLANKFINAL;  
...  
BLANKFINAL = 0;
```

□

☞ Namen von Konstanten sollten nur Großbuchstaben enthalten. Das Unterstrich-Symbol „\_“ kann verwendet werden, um unterschiedliche Wörter in einem Konstantennamen zu trennen.

**Selbsttestaufgabe 10.2-1:**

Seien

```
double height;  
double base;
```

zwei Variablen mit beliebigen Werten, die die Länge der Höhe bzw. Grundseite eines Dreiecks darstellen. Deklarieren Sie die Variable `area` und speichern Sie in ihr den Betrag des Flächeninhalts des Dreiecks.

**Selbsttestaufgabe 10.2-2:**

Überprüfen Sie die nachstehende Folge von Initialisierungen und Zuweisungen und erklären Sie für jede Zeile, welchen Effekt die Auswertung der Ausdrücke auf die drei Variablen `x`, `y` und `z` hat.

```
int x = 5;           // 1  
int y = 4;           // 2  
int z = 3;           // 3  
x = x - 1;           // 4  
y = z + 1;           // 5  
z = x + y + z;       // 6
```

**Selbsttestaufgabe 10.2-3:**

Formulieren Sie die folgenden mathematischen Ausdrücke als Java-Anweisungen. Alle Variablen sollen vom Typ `double` sein. Nehmen Sie vereinfachend an, dass  $\pi$  den Wert 3,14 hat.

a)

$$s = \frac{1}{2}gt^2$$

b)

$$A = \pi r^2$$

c)

$$V = \frac{\pi h}{3}(r_1^2 + r_1 r_2 + r_2^2)$$

Inkrementoperator  
Dekrementoperator

Die einstelligen Inkrement- und Dekrementoperatoren

`var++`, `var--`, `++var` und `--var`,

erhöhen bzw. erniedrigen eine Variable `var` um den Wert 1. [JLS: § 15.4, § 15.5]  
Nach der Ausführung von

```
int hoehe = 10;
hoehe++;
```

hat `hoehe` den Wert 11.

Nebeneffekt  
Seiteneffekt

Werden Inkrement- oder Dekrementoperatoren auf der rechten Seite einer Zuweisung eingesetzt, so entfalten sie einen sogenannten Neben- oder Seiteneffekt. So erhalten durch die Ausführung von

```
int a = 3;
int b = ++a;
```

beide Variablen `a` und `b` den Wert 4.

Der semantische Unterschied zwischen den Ausdrücken `var++` und `++var` (Entsprechendes gilt für `var--` und `--var`) besteht darin, dass im Ausdruck `var++` als Ergebnis der Wert der Variablen `var` vor der Erhöhung geliefert wird, während im Ausdruck `++var` der Wert von `var` nach dem Hochzählen geliefert wird. Nach der Ausführung von

```
int a = 3;
int b = a++;
```

hat `a` den Wert 4, `b` jedoch den Wert 3, da für die Zuweisung an `b` der Wert von `a` vor dem Hochzählen geliefert wurde.

**Selbsttestaufgabe 10.2-4:**

Aus welcher der folgenden Anweisungen resultiert  $n == 1$  und  $x == 1$ , wenn zuvor jeweils

```
int n = 0;
int x = 1;
```

deklariert wurde?

a)

```
n = x++ + x++;
```

b)

```
n = n++ - x++;
```

c)

```
n = x-- + -x++;
```

**Selbsttestaufgabe 10.2-5:**

Entwickeln Sie einen Ausdruck, der genau dann zu `true` evaluiert, wenn `x` einen Wert kleiner 10 oder einen geraden Wert größer 20 gespeichert hat.

**Selbsttestaufgabe 10.2-6:**

Entwickeln Sie einen Ausdruck, der genau dann zu `true` evaluiert, wenn `x` ein Teiler von `y` ist.

# 11 Typanpassung

In der Mathematik sind wir es gewohnt, gemischte Ausdrücke wie

$31,51 + 12$

zu schreiben, und wir erwarten als Ergebnis den Wert 43,51. In Java erwarten zwei-stellige Operatoren jedoch zunächst, dass ihre Operanden vom gleichen Typ sind.

Ist dies nicht der Fall, kann in gewissen Fällen durch den Java-Übersetzer eine  
 Typanpassung Typanpassung implizit (automatisch) durchgeführt werden, um die Operandentypen  
 in Übereinstimmung zu bringen. Darüber hinaus kann eine Typanpassung (synonym  
 Typumwandlung Typumwandlung) explizit erzwungen werden.

## 11.1 Implizite Typanpassung

implizite Typanpassung Numerische Typen weisen gewisse Familienbeziehungen in dem Sinne auf, dass be-  
 stimmte Typen im Wertebereich anderer Typen enthalten sind. Die Werte vom Typ  
`byte` zum Beispiel sind im Wertebereich des Typs `int` enthalten. Da Java jedoch  
 eine stark typisierte Sprache ist, muss jeder Wert mit einem Datentyp verbunden  
 sein, so dass der `byte`-Wert 23 vom `int`-Wert 23 unterschieden wird.

Typenerweiterung Durch die Erweiterung (engl. widening) einer Zahl des Typs  $t$  wird diese in eine  
 Zahl des Typs  $u$  umgewandelt. Der Wertebereich des Typ  $t$  muss eine Teilmenge  
 des Wertebereichs des Typs  $u$  sein.

Der Java-Übersetzer wendet solche Erweiterungsanpassungen automatisch an, falls  
 eine Typverträglichkeit (engl. type compatibility) zwischen ungleichen Typen ge-  
 geben ist.

### Definition 11.1-1: Typverträglichkeit

Typverträglichkeit Ein Datentyp  $t$  ist verträglich mit einem anderen Datentyp  $u$ , abgekürzt:

$$t < u$$

*falls die Werte des Typs  $t$  in Werte vom Typ  $u$  ausgeweitet werden können.*

□

Der Java-Übersetzer erlaubt zwischen numerischen Werten die in Tab. 11.1-1 ange-  
 gebenen Erweiterungsanpassungen. [JLS: § 5.1.2]

**Tab. 11.1-1:** Erweiterungsanpassungen

von	nach
byte	short, int, long, float <b>oder</b> double
short	int, long, float <b>oder</b> double
int	long, float <b>oder</b> double
long	float <b>oder</b> double
float	double

**Beispiel 11.1-1:**

*Während der Auswertung des Ausdrucks*

```
31.51 + 12
```

*wird die Zahl 12 (vom Typ `int`) in die Zahl `12.0` vom Typ `double` umgewandelt. Als Ergebnis erhalten wir den Wert `43.51` vom Typ `double`.*

□

**Bemerkung 11.1-1:**

*Beachten Sie, dass Typumwandlungen nur während der Auswertung eines Ausdrucks und nur auf Werten vorgenommen werden. Der Typ einer Variablen ändert sich hierbei nicht.*

*Hätten wir im vorherigen Beispiel anstelle der Zahl 12 eine `int`-Variable `a` verwendet, deren aktueller Wert 12 ist, so wäre diese nach Auswertung des Ausdrucks*

```
31.51 + a
```

*unverändert eine `int`-Variable mit dem Wert 12.*

□

Auch der Zuweisungsoperator verlangt die Typverträglichkeit zwischen dem Ausdruck auf der rechten Seite einer Anweisung und der Variablen auf der linken Seite. Der Typ des Ausdrucks muss implizit in den Typ der Variablen umgewandelt werden können. Ist keine Typverträglichkeit gegeben, weist der Java-Übersetzer das Programm zurück.

**Beispiel 11.1-2:**

*Bei der Zuweisung eines `int`-Werts an eine `double`-Variable*

```
double x;
...
x = 3;
```

*wird der `int`-Wert 3 vor der Zuweisung in den `double`-Wert `3.0` umgewandelt.*

*Versuchen wir hingegen, eine `int`-Variable `y` mit einem `double`-Wert `3.0` zu belegen, so erhalten wir vom Übersetzer eine Fehlermeldung, da keine Typverträglichkeit vorliegt:*

```

possible loss of precision
found    : double
required: int
y  = 3.0;
    ^
1 error

```

□

Wie wir bereits wissen, werden Zuweisungen von rechts nach links ausgewertet.

### Definition 11.1-2: Auswertung einer Zuweisung

Auswertung einer  
Zuweisung

Eine Zuweisung der Form „ $v = e$ ;“ wird in folgenden Schritten ausgewertet:

1. Der Typ der Variablen  $v$  wird ausgewertet.
2. Der Wert  $w(e)$  des Ausdrucks  $e$  wird ausgewertet.
3. Der berechnete Wert wird der Variablen zugewiesen, vorausgesetzt, dass der Typ des Ausdrucks  $e$  mit dem Typ der Variablen  $v$  verträglich ist.

Falls die Typen des Ausdrucks und der Variablen nicht identisch aber miteinander verträglich sind, wird der Typ des Ausdrucks in den Typ der Variablen umgewandelt.  
[JLS: § 5.2]

□

Aus Definition 11.1-2 folgt, dass der Typ der Variablen auf der linken Seite des „ $=$ “-Operators die Auswertung des Ausdrucks auf der rechten Seite nicht beeinflusst. Eine mögliche Typanpassung erfolgt erst am Ergebnis der Auswertung. Die Nichtbeachtung dieser Reihenfolge stellt eine häufige Fehlerquelle dar, wie die folgenden beiden Beispiele illustrieren.

### Beispiel 11.1-3: Division

Nach Ausführung der Initialisierung

```
double x = 1 / 4;
```

hat die Variable  $x$  nicht, wie vermutlich beabsichtigt, den Wert  $0.25$ , sondern den Wert  $0.0$ . Warum? Auf der rechten Seite wurde zunächst eine Division zweier `int`-Werte ausgeführt. Das Ergebnis einer `int`-Division in Java ist ein `int`-Wert, wobei der Divisionsrest ignoriert wird (vgl. Abschnitt 9.3). Die Auswertung der rechten Seite ergibt somit den `int`-Wert  $0$ . Dieser wird anschließend, um der `double`-Variablen  $x$  zugewiesen werden zu können, in den `double`-Wert  $0.0$  umgewandelt.

Um zu erreichen, dass  $x$  den Wert  $0.25$  erhält, hätte mindestens einer der Werte auf der rechten Seite in Gleitpunkt-Notation geschrieben sein müssen. So bewirkt die Initialisierung

```
double x = 1.0 / 4;
```

*dass bereits vor der Auswertung der Division der `int`-Wert 4 in den `double`-Wert 4.0 umgewandelt wird, die anschließende `double`-Division das Ergebnis 0.25 vom Typ `double` liefert und dieses (nun ohne weitere Umwandlung) der Variablen `x` zugewiesen wird.*

□

#### **Beispiel 11.1-4: Überschreiten des Wertebereichs**

*Nach der Ausführung von*

```
int a = 2000000;
int b = 7000000;
long z = a * b;
```

*ist in `z` der Wert -1593384960 gespeichert und nicht, wie es mathematisch korrekt wäre, 14000000000000, obwohl der Wertebereich von `z` diesen Wert zulassen würde. Das Ergebnis einer `int`-Multiplikation ist ein `int`-Wert. Die `int`-Multiplikation resultiert in unserem Beispiel in einem sogenannten Überlauf (engl. *overflow*), der bei einer Überschreitung des Wertebereichs auftritt.*

Überlauf

*Eine Erweiterung des Wertebereichs hätte bereits vor der Multiplikation stattfinden müssen. Wäre mindestens eine der beiden Variablen `a` und `b` vom Typ `long`, so erhielten wir für `z` den erwarteten Wert 14000000000000:*

```
int a = 2000000;
long b = 7000000;
long z = a * b;
```

*Nun wird der Wert von `a` implizit als ein Wert vom Typ `long` aufgefasst und es wird eine `long`-Multiplikation ausgeführt.*

□

*Während ganzzahlige Typerweiterungen niemals zu einem Informationsverlust führen, kann die Erweiterung von ganzen Zahlen in Gleitkommawerte eine Verminderung der Genauigkeit bewirken. Zwar wird der Wertebereich erweitert, doch können bei der Umwandlung eines `int` oder `long` zu `float` oder `double` einige der weniger signifikanten Ziffern verloren gehen.*

Verminderung der Genauigkeit

#### **Beispiel 11.1-5:**

*Nach Ausführung der Initialisierung*

```
float b = 1000000027;
```

*mit impliziter Typerweiterungen von `int` nach `float` enthält die Variable `b` den Wert 1.0E9.*

□

*Java unterstützt auch eine Typerweiterung von `char`-Werten zu Werten vom Typ `int`, `long`, `float` oder `double`. Wie bei der Erweiterung numerischer Werte wird die Erweiterung von Zeichen in numerische Werte vom Java-Übersetzer automatisch angewandt, wenn dies für die Typenverträglichkeit notwendig ist.*

**Beispiel 11.1-6:**

Nach der Ausführung von

```
int x = 'A' + 32;
```

hat  $x$  den Wert 97. 'A' wurde zuvor in den `int`-Wert 65 umgewandelt.

□

**Bemerkung 11.1-2: Unicode-Nummerierung von Zeichen**

Die Erweiterung von Zeichen in numerische Werte folgt der Unicode-Nummerierung der Zeichen. Die Großbuchstaben 'A' bis 'Z' korrespondieren mit den Werten 65 bis 90, die Kleinbuchstaben 'a' bis 'z' mit den Werten 97 bis 122. Die Differenz zwischen zwei alphabetisch gleichen Klein- bzw. Großbuchstaben (z. B. 'a' - 'A') beträgt immer 32.

Mit Hilfe der Typumwandlung lässt sich somit feststellen, ob zwei Zeichen alphabetisch gleich sind. Seien die `char`-Variable `kleinBuchstabe` mit irgendeinem Kleinbuchstaben und die `char`-Variable `grossBuchstabe` mit einem beliebigen Großbuchstaben belegt. Nach der Ausführung von

```
boolean alphabetischGleich =
    (kleinBuchstabe - grossBuchstabe) == ('a' - 'A');
```

enthält die Variable `alphabetischGleich` den Wert `true`, falls beispielsweise `kleinBuchstabe` und `grossBuchstabe` mit 'd' und 'D' belegt waren, oder den Wert `false`, falls beispielsweise `kleinBuchstabe` und `grossBuchstabe` mit 'd' und 'M' belegt waren.

□

**Selbsttestaufgabe 11.1-1:**

Entwickeln Sie einen Ausdruck, der genau dann `true` liefert, wenn die beiden Buchstaben `buchstabe1` und `buchstabe2` identisch sind oder sich nur in Groß- und Kleinschreibung unterscheiden.

**Selbsttestaufgabe 11.1-2:**

Welcher der folgenden Ausdrücke kann einer Variablen `b` vom Typ `byte` zugewiesen werden, und welchen Wert nimmt `b` nach Auswertung der Anweisung an?

```
3*2
129
5+'a'
12.5
78
```



**Selbsttestaufgabe 11.1-3:**

Welche Werte haben die Variablen *x*, *y* und *erg* in den folgenden Anweisungen nach deren Ausführung. Überprüfen Sie das Ergebnis anschließend, indem Sie den Quelltext ausführen.

```
int x = 010;
double y = 2.5;
double erg = x + y;
x++;
```

## 11.2 Explizite Typanpassung

Es gibt Situationen, in denen man eine Typenumwandlung erzwingen will, die nicht automatisch (implizit) stattfindet. Die erzwungene Typumwandlung (engl. type casting) erlaubt es, Typumwandlungen explizit anzugeben.

explizite Typanpassung

### Definition 11.2-1: Erzwungene Typumwandlung

Java unterstützt eine erzwungene Typumwandlung, um den Typ eines numerischen Werts während der Laufzeit in einen numerischen Typ ähnlichen Werts zu ändern. Die Syntax einer erzwungenen Typumwandlung ist:

erzwungene  
Typumwandlung

```
(typeName) expression
```

wobei *typeName* ein primitiver Typ und *expression* ein Ausdruck ist. Die Typumwandlung wird von rechts nach links durchgeführt und liefert einen Wert vom Typ *typeName*, vorausgesetzt, dass die Konversion keine Kompatibilitätsregeln verletzt. [JLS: § 5.5, § 15.16]

□

Typanpassungen fallen in zwei grundlegende Kategorien: Erweiterung (engl. widening) und Einengung (engl. narrowing). Erweiterungen können sowohl implizit ausgeführt, als auch explizit erzwungen werden. Die im vorherigen Abschnitt vorgestellten Gesetzmäßigkeiten impliziter Erweiterungen gelten auch für explizite Erweiterungen.

### Beispiel 11.2-1: Division

Das Divisionsproblem aus Beispiel 11.1-3 lässt sich auch mit einer expliziten Erweiterung lösen:

```
double x = (double) 1 / 4;
```

□

**Beispiel 11.2-2: Überschreiten des Wertebereichs**

Das in Beispiel 11.1-4 gezeigte Problem „Überschreiten des Wertebereichs“ kann auch mit einer expliziten Erweiterung einer der zu multiplizierenden `int`-Variablen `a` und `b` gelöst werden.

```
int a = 2000000;
int b = 7000000;
long z = a * (long) b;
```

□

Typeinengung  
cast-Operator

Einengungen werden vom Java-Übersetzer nicht automatisch angewandt, sondern können nur mit Hilfe des cast-Operators (`typeName`) erzwungen werden. Sie wandeln Werte eines Datentyps `u` in Werte eines anderen Typs `t` um, wobei der Wertebereich des Typs `t` keine Teilmenge des Wertebereichs des Typs `u` ist. Einengende Typenumwandlungen können deshalb einen Verlust an Information hervorrufen.

Die Auswertung des cast-Operators erfolgt unmittelbar nach der Auswertung des zugehörigen Ausdrucks (vgl. Tab. 9.4-1).

Java erlaubt es uns, zwischen numerischen Typen beliebige einengende Umwandlungen zu erzwingen. [JLS: § 5.1.3]

**Beispiel 11.2-3:**

*Nach Ausführung der drei Initialisierungen*

```
double genau = 12345.6 / 7.8;
int ganzzahlig = (int) genau;
byte uebergelaufen = (byte) ganzzahlig;
```

*hat die Variable `genau` den Wert 1582.769230769231, die Variable `ganzzahlig` den Wert 1582 und die Variable `uebergelaufen` den Wert 46. Wenn eine Gleitkommazahl in eine ganze Zahl umgewandelt wird, so wird nicht gerundet, sondern die Nachkommastellen werden abgeschnitten.*

□

Wie bei allen Operatoren können wir auch beim cast-Operator durch Klammerung die Auswertungsreihenfolge beeinflussen.

**Beispiel 11.2-4: Geburtenrate**

*Mit der Geburtenrate wird die durchschnittliche Anzahl der Geburten pro Jahr und 1000 Einwohner angegeben. Wir berechnen die innerhalb eines Jahres erwartete (ganzzahlige) Geburtenzahl in einer Gemeinde. Zur Berechnung wollen wir die exakte (Gleitkomma-)Geburtenrate verwenden:*

```
double geburtenrate = 8.25;
int einwohner = 123456;
int erwarteteGeburten =
    (int) (einwohner * geburtenrate / 1000);
```

Durch die Klammerung erreichen wir, dass erst das Ergebnis der Berechnung (die zuvor mit Gleitkomma-Genauigkeit durchgeführt wurde) auf einen ganzzahligen Wert eingeengt wird.

□

### Selbsttestaufgabe 11.2-1:

Gegeben seien die folgenden Berechnungen des Jahreszins auf Basis einer `long`-Variable, die den Betrag in Cent speichert. Diese Berechnungen ist fehlerhaft. Finden Sie heraus warum und beheben Sie den Fehler.

```
long betragInCent = 2342352;
int prozentsatz = 3;
long jahreszinsInCent = prozentsatz / 100 * betragInCent;
```

Auch einengende Umwandlungen von allen numerischen Typen in den Typ `char` sowie von `char` nach `byte` oder `short` sind möglich.

### Beispiel 11.2-5: Umwandlung von `int`-Werten in `char`-Werte

Der Ausdruck

```
(char) 123
```

erzwingt eine Umwandlung des `int`-Werts 123 in das Zeichen '{', das mit dem Ganzzahlwert 123 in Unicode codiert ist.

□

### Beispiel 11.2-6: Ableitung eines Großbuchstabens

Seien `grossBuchstabe` und `kleinBuchstabe` zwei Variablen vom Typ `char`, `kleinBuchstabe` sei mit einem Kleinbuchstaben initialisiert. Für die numerische Differenz zwischen gleichen Klein- und Großbuchstaben deklarieren wir eine Konstante:

```
final int ABSTAND = 'a' - 'A';
```

Bei der Zuweisung

```
grossBuchstabe = (char) (kleinbuchstabe - ABSTAND);
```

erfolgt zunächst eine implizite Erweiterung des Wertes von `kleinBuchstabe` nach `int`. Das `int`-Ergebnis der Subtraktion (beachten Sie die Klammern) wird explizit eingeengt auf den Typ `char` und kann anschließend der Variablen `grossBuchstabe` zugewiesen werden.

□

**Bemerkung 11.2-1: Typeinengung bei char-Werten**

*Es verwundert zunächst, dass die Typumwandlung zwischen byte- oder short-Werten einerseits und char-Werten andererseits in beide Richtungen einengend, d. h. verlustbehaftet ist. Dies ist dadurch zu erklären, dass char-Werte nicht durch ein Vorzeichen gekennzeichnet sind. Der Wertebereich eines char-Wertes repräsentiert die Unicode-Nummerierung 0 . . 65535, so dass sich die Wertebereiche von byte oder short (vgl. Tab. 9.1-1) und char nur überlappen, aber nicht überdecken.*

□

Kompatibilitätsregeln

Die mit Hilfe des cast-Operators möglichen Typumwandlungen fassen wir in folgenden Kompatibilitätsregeln (engl. compatibility rules) für einfache Typen zusammen:

- Jeder Typ kann in sich selbst umgewandelt werden.
- Jeder numerische Typ kann in jeden anderen numerischen Typen umgewandelt werden.
- Typ char kann in jeden numerischen Typen umgewandelt werden und jeder numerische Typ kann in Typ char umgewandelt werden.
- Typ boolean kann in keinen anderen Typ umgewandelt werden und kein andere Typ kann in Typ boolean umgewandelt werden.

**Selbsttestaufgabe 11.2-2:**

*Das Verfahren, um eine Temperaturangabe in Grad Fahrenheit in die Einheit Celsius umzuwandeln, lautet: Temperatur in °C = 5 / 9 \* (Temperatur in °F - 32).*

*Gegeben seien zwei int-Variablen celsius und fahrenheit, letztere sei initialisiert. Schreiben sie eine Zuweisung, durch die die Variable celsius die korrekte (auf die nächste Ganzzahl abgerundete) Temperaturangabe erhält.*

**Selbsttestaufgabe 11.2-3:**

*Untersuchen Sie die folgenden Initialisierungen. Werden die Zuweisungen vom Java-Übersetzer akzeptiert? Falls nein: warum nicht? Falls ja: entsteht ein Informationsverlust?*

```
int anzahl = (byte) 32;
double preis = (byte) anzahl * 123.45;
boolean bezahlt = (boolean) true;
```

**Selbsttestaufgabe 11.2-4:**

*Entwickeln Sie ein Programm, das in der Variablen `durchschnitt` vom Typ `double` den Durchschnitt der drei Variablen `a`, `b` und `c` vom Typ `long` berechnet.*

## 12 Anweisungen

Anweisung  
einfache Anweisung

Anweisungen (engl. statement) sind die grundlegenden Programmierkonstrukte, aus denen ein Programm aufgebaut ist. Einfache Anweisungen haben wir bereits kennen gelernt: Ausdrücke, die mit einem Semikolon enden, sind Anweisungen. Jede Deklaration ist eine Anweisung. Oft sind Zuweisungen eigenständige Anweisungen.

In der ersten Kurseinheit haben wir Aktivitätsdiagramme zur Beschreibung von Verfahrensabläufen kennen gelernt. In Java formulieren wir Verfahrensabläufe, indem wir Anweisungen auf vielfältige Weise gruppieren und kombinieren.

Sequenz

Aneinander gereiht Anweisungen gestatten es uns, einfache lineare Abfolgen (Sequenzen) im Programmgeschehen zu formulieren. Anweisungssequenzen haben wir in dieser Kurseinheit schon mehrfach verwendet.

Oft hängt aber der Verlauf des Geschehens von bestimmten Bedingungen ab. Wenn z. B. in unserem Blumenladen eine Kundin mit Kreditkarte bezahlen möchte, muss sich das System anders verhalten als bei einer Barzahlung.

Um die Auswahl eines bestimmten Elements aus einer größeren Menge gleichartiger Daten zu ermöglichen, etwa die Wahl bestimmter Lieferanten, benötigen wir Sprachkonstrukte, mit deren Hilfe die Einträge systematisch durchwandert und mit vorgegebenen Auswahlkriterien verglichen werden können.

Um solche anspruchsvolleren Verfahrensabläufe zu formulieren, benötigen wir Ausdrucksmittel, die es uns erlauben, Wiederholungen oder Entscheidungen auszudrücken. Auch diese Sprachkonstrukte werden in Programmiersprachen als Anweisungen bezeichnet.

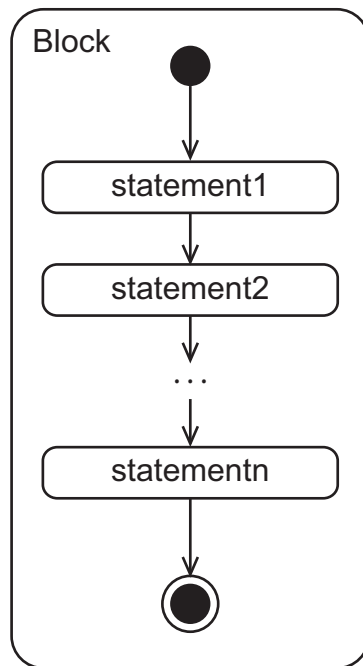
### 12.1 Blöcke

Anweisungen können zu Anweisungsblöcken aneinander gereiht werden. Blöcke helfen dabei, ein Programm zu strukturieren.

#### Definition 12.1-1: Block

Block

*Eine Folge von Anweisungen, die in geschweiften Klammern „{“ und „}“ eingefasst sind, nennt man einen Block. Der Aufbau und die Abarbeitung eines Blocks werden schematisch im folgenden Aktivitätsdiagramm dargestellt. Die Anweisungen eines Blocks werden von oben nach unten sequenziell abgearbeitet. [JLS: § 14.2]*



**Abb. 12.1-1:** Struktur und Semantik eines Blocks.

□

Ein Block kann beliebig viele (auch null) Anweisungen enthalten. Ein Block ist eine Anweisung.

### Beispiel 12.1-1: Block

Der folgende Programmauszug gibt ein Beispiel für einen Block:

```

{
    int bestellteTulpen = 12;
    int bestellteRosen = 16;
    double preisTulpen = 0.80;
    double preisRosen = 2.10;
    double gesamtpreis = bestellteTulpen * preisTulpen
                        + bestellteRosen * preisRosen;
}
  
```

□

☞ In Blöcken sollten Einrückung verwendet werden. Die gebräuchliche Einheit für Einrückungen sind vier Leerzeichen. Wir nennen dies die 4-Leerzeichen-Regel (engl. 4-space rule).

☞ Jede Zeile innerhalb eines Blocks sollte höchstens eine Anweisung enthalten.

☞ Eine Zeile sollte nicht länger als 80 Zeichen sein. Falls eine Anweisung nicht in eine einzelne Zeile passt, trennen Sie sie entsprechend den folgenden allgemeinen Trennungsregeln in mehrere Zeilen auf:

- Trennen Sie nach einem Komma.
- Trennen Sie vor einem Operator.
- Richten Sie die neue Zeile so aus, dass der Beginn des Ausdrucks auf derselben Höhe beginnt wie der Ausdruck in der Zeile davor.

## 12.2 Kontrollstrukturen

Kontrollstrukturen Java bietet drei Arten von Kontrollstrukturen an, mit denen der Ablauf eines Programms formuliert werden kann: Bedingungs-, Wiederholungs- und strukturierte Sprunganweisungen.

Bedingungsanweisungen dienen dazu, zwischen alternativen Anweisungen auszuwählen.

Mit Wiederholungsanweisungen kann man die wiederholte Abarbeitung eines Blocks beschreiben. Die Iteration wird in Abhängigkeit vom jeweiligen Wert eines booleschen Ausdrucks wiederholt oder beendet.

Strukturierte Sprunganweisungen werden benötigt, um die Ausführung einer Folge von Anweisungen zu unterbrechen und den Kontrollfluss an anderer Stelle im Programm fortzuführen.

Diese Anweisungsarten werden in den folgenden Kapiteln vorgestellt.



## 13 Bedingungsanweisungen

Viele Algorithmen hängen von der Möglichkeit ab, bei der Ausführung zwischen alternativen Gruppen von Anweisungen abhängig von einer Bedingung auszuwählen. Java unterstützt zwei Arten von Verzweigungsanweisungen, die einfache Fallunterscheidung oder `if`-Anweisung und die Mehrfach-Fallunterscheidung oder `switch`-Anweisung.

### 13.1 Die einfache Fallunterscheidung

Die einfache Fallunterscheidung oder `if`-Anweisung (engl. `if statement`) ermöglicht die Auswahl zwischen zwei Anweisungen (die auch Blöcke sein können).

einfache  
Fallunterscheidung  
`if`-Anweisung

#### Definition 13.1-1: `if`-Anweisung

Die `if`-Anweisung hat die folgende Form:

```
if (expression)
    statement1
else
    statement2
```

wobei

- *expression* ein boolescher Ausdruck sein muss und
- *statement1* und *statement2* Anweisungen sind.

Während der Ausführung des Programms wird eine `if`-Anweisung wie folgt ausgewertet:

- Zuerst wird der boolesche Ausdruck *expression* ausgewertet. Wenn das Ergebnis `true` liefert, wird *statement1* ausgeführt, und der Kontrollfluss wird hinter der `if`-Anweisung fortgesetzt.
- Wenn die Auswertung von *expression* den Wert `false` liefert, wird *statement1* ignoriert und stattdessen *statement2* ausgeführt; danach wird die `if`-Anweisung verlassen. [JLS: § 14.9]

Die Semantik einer `if`-Anweisung wird in Abb. 13.1-1 veranschaulicht:

Der Alternativzweig kann bei einer `if`-Anweisung auch weggelassen werden, falls nur bei positiver Auswertung des Bedingungsausdrucks *expression* eine spezielle Anweisung ausgeführt werden soll:

```
if (expression)
    statement
```

Die Semantik dieser Form der `if`-Anweisung ist in Abb. 13.1-2 dargestellt.

□

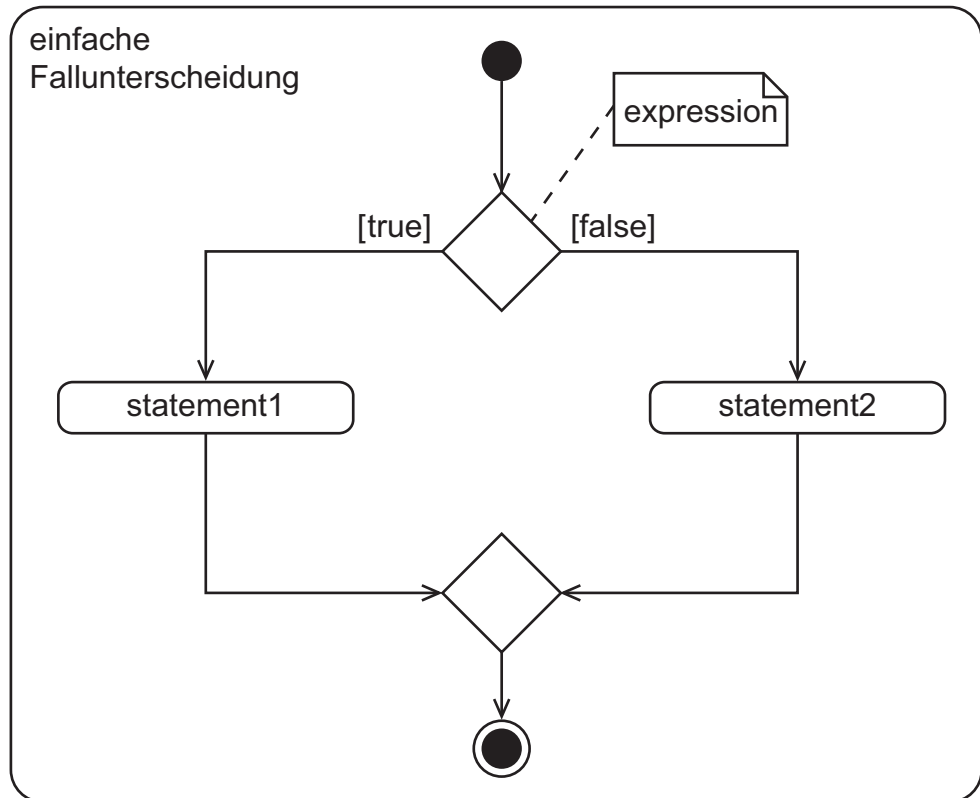


Abb. 13.1-1: Struktur und Auswertung einer if-Anweisung

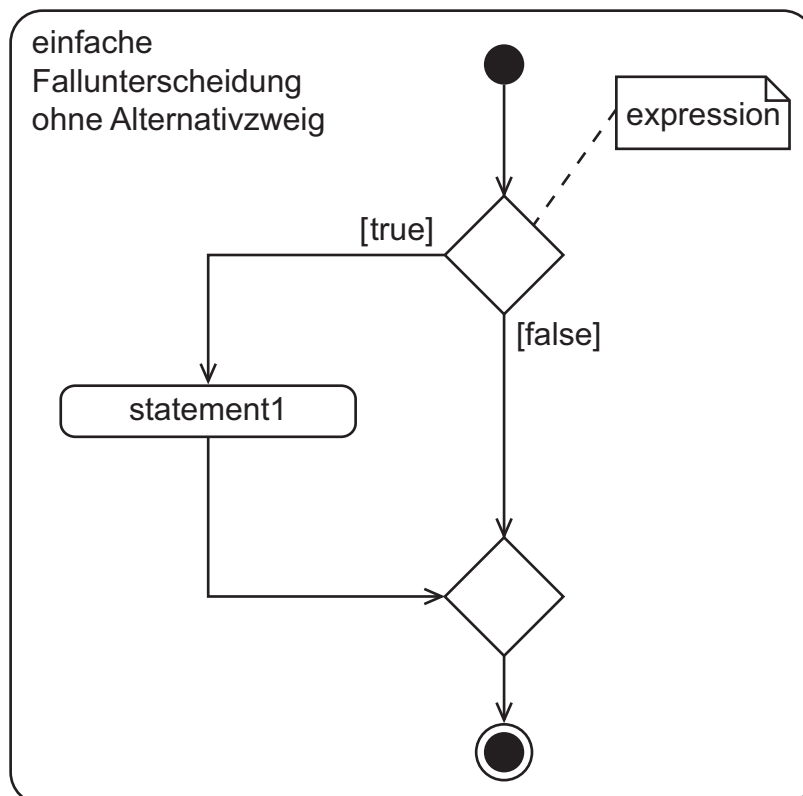


Abb. 13.1-2: if-Anweisung ohne Alternativzweig

**Beispiel 13.1-1: if-Anweisung**

Mit der folgenden Anweisungsfolge können wir den Absolutwert einer ganzzahligen Variablen errechnen:

```
int absValue;
int x;
...
if (x >= 0)
    absValue = x;
else
    absValue = -x;
```

□

Einfache Fallunterscheidungen können auch verschachtelt sein. Dies bedeutet, dass der Rumpf einer `if`-Anweisung andere `if`-Anweisungen beinhalten kann.

Bei verschachtelten `if`-Anweisungen wird, falls keine Zuordnung der Alternativzweige durch entsprechende Blockbildung vorgegeben ist, ein Alternativzweig „`else statement`“ immer der unmittelbar vorangehenden `if`-Anweisung zugeordnet.

☞ Um derartige Konfusionen bei verschachtelten `if`-Anweisungen zu vermeiden, empfehlen wir dringend, die Zweige einer `if`-Anweisung als Block zu schreiben, auch wenn der Block nur eine Anweisung enthält:

```
if (expression) {
    statement11
    statement12
    ...
    statement1m
} else {
    statement21
    statement22
    ...
    statement2n
}
```

**Beispiel 13.1-2: Verschachtelte if-Anweisungen**

```
if (x >= 0) {
    if (x != 0) {
        absValue = x;
    }
} else {
    absValue = -x;
}
```

*Durch die Klammerung ist der Alternativzweig der ersten (äußeren) if-Anweisung zugeordnet. Die äußere if-Anweisung enthält eine weitere if-Anweisung ohne Alternativzweig.*

□

**Selbsttestaufgabe 13.1-1:**

*Implementieren Sie die Berechnung aus Selbsttestaufgabe 4.2-1 in Java.*

**Selbsttestaufgabe 13.1-2:**

*Gegeben seien die beiden Variablen*

```
boolean istGrossbuchstabe;  
char testZeichen = 'F';
```

*Schreiben Sie eine if-Anweisung, die prüft, ob es sich bei testZeichen um einen Großbuchstaben handelt. Das Ergebnis der Prüfung soll in istGrossbuchstabe gespeichert werden. Für testZeichen können Sie auch ein anderes Zeichen wählen bzw. verschiedene Zeichen ausprobieren.*

**Lösungshinweis:** *Um das Ergebnis auf dem Bildschirm sichtbar zu machen, können Sie nach der if-Anweisung noch folgende Zeile anfügen:*

```
System.out.println(istGrossbuchstabe);
```

**Selbsttestaufgabe 13.1-3:**

*Zeichnen sie ein Aktivitätsdiagramm, das den Kontrollfluss der verschachtelten if-Anweisung aus Beispiel 13.1-2 veranschaulicht.*

**Selbsttestaufgabe 13.1-4:**

Warum sind die folgenden Anweisungen fehlerhaft?

a)

```
int x = 10;
int y = 2;
if (x = 1) {
    y++;
}
```

b)

```
int x = 10;
int y = 10;
if (x && y == 10) {
    x++;
}
```

c)

```
int x = 3;
if x > 2 then x++;
```

**Selbsttestaufgabe 13.1-5:**

Gegeben seien zwei Variablen *start* und *ende* vom Typ *int*, die eine Uhrzeit repräsentieren. So steht 800 für 8 Uhr und 1645 für 16:45 Uhr. Entwickeln Sie ein Programm, das die Stunden und Minuten zwischen den beiden Zeiten berechnet. Bei *start* = 800 und *ende* = 1645 soll das Programm somit

```
8 Stunden
45 Minuten
```

ausgeben. Bei *start* = 1645 und *ende* = 1500 soll das Programm

```
22 Stunden
15 Minuten
```

ausgeben. Sie können davon ausgehen, dass nur gültige Uhrzeiten verwendet werden.

## 13.2 Die Mehrfach-Fallunterscheidung

Mehrfach-Fallunterscheidung

Eine `if`-Anweisung dient dazu, aus zwei Anweisungen genau eine auszuwählen und auszuführen. Manchmal stehen wir aber vor der Aufgabe, zwischen mehr als zwei bedingten Anweisungsalternativen auszuwählen. Der Gebrauch von verschachtelten `if`-Anweisungen kann dabei oft zu unüberschaulichen Lösungen führen. Für diesen Fall bietet Java in Form der `switch`-Anweisung eine bessere Möglichkeit an. Mit ihr kann man eine Reihe von Werten, die jeweils eine alternative Anweisung auswählen, abarbeiten.

`switch`-Anweisung

Javas `switch`-Anweisung unterstützt eine Fallunterscheidung zwischen mehreren alternativen Anweisungen. Die `switch`-Anweisung besteht aus einer Sammlung von alternativen Werten und zugeordneten Anweisungen.

### Definition 13.2-1: `switch`-Anweisung

Die `switch`-Anweisung hat die folgende Form:

```
switch (expression) {
  case k1:
    statement1;
  case k2:
    statement2;
  ...
  case kn:
    statementn;
  default:
    statementd;
}
```

wobei

- der Ausdruck *expression* vom Typ *byte*, *short*, *int* oder *char* sein muss;
- alle *k1*, ..., *kn* Konstanten des gleichen Datentyps wie *expression* oder eines damit verträglichen Typs sein müssen;
- *statement1*, ..., *statementn* und *statementd* Anweisungen sind.

Während der Programmausführung wird eine `switch`-Anweisung wie folgt ausgewertet:

- Zuerst wird *expression* ausgewertet. Falls das Ergebnis dieser Auswertung ein Wert ist, der mit einer der Konstanten *ki* für  $i = 1, \dots, n$  übereinstimmt, dann werden die zugehörige Anweisung *statement<sub>i</sub>* und alle nachgeordneten Anweisungen *statement<sub>i+1</sub>* bis *statement<sub>n</sub>* ausgeführt.

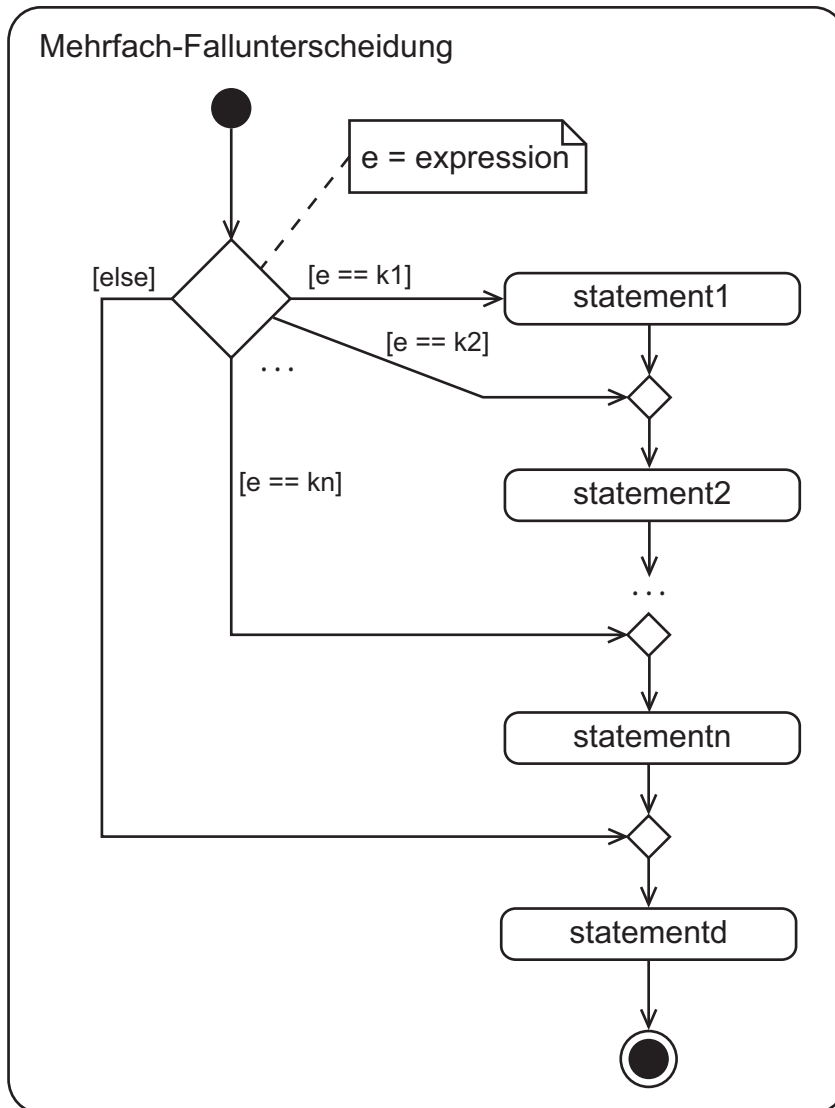
Standardfall

- Falls ein Standardfall (engl. *default case*) angegeben ist, wird die zugehörige Anweisung *statement<sub>d</sub>* abgearbeitet und danach die `switch`-Anweisung verlassen. *statement<sub>d</sub>* wird auch abgearbeitet, wenn das Ergebnis der Auswertung von *expression* mit keiner der Konstanten *ki* übereinstimmt. Ist

*kein Standardfall angegeben, so wird in diesem Fall keine der Anweisungen der Fallunterscheidung ausgeführt. [JLS: § 14.11]*

*Diese Semantik ist in Abb. 13.2-1 grafisch dargestellt.*

□



**Abb. 13.2-1:** Mehrfach-Fallunterscheidung

**Bemerkung 13.2-1:**

*Die geschweiften Klammern sind ein zwingender Bestandteil der switch-Anweisung.*

□

**Beispiel 13.2-1: Mehrfach-Fallunterscheidung**

*Die Wirkung des folgenden Beispiels:*

```
int weekdays;
...
switch (weekdays) {

case 1:
    System.out.println("Monday");
case 2:
    System.out.println("Tuesday");
case 3:
    System.out.println("Wednesday");
case 4:
    System.out.println("Thursday");
case 5:
    System.out.println("Friday");
case 6:
    System.out.println("Saturday");
case 7:
    System.out.println("Sunday");
default:
    System.out.println("there are no more weekdays");
}
```

*ist, dass der Wert der Variablen `weekdays` bei der Ausführung der Fallunterscheidung darüber entscheidet, welche Tage der Woche angezeigt werden, wenn die `switch`-Anweisung ausgeführt wird. Falls `weekdays` den Wert 1 hat, werden alle Wochentage von Monday bis Sunday und der Text „there are no more weekdays“ angezeigt, jeweils in einer separaten Zeile.*

*Falls `weekdays` den Wert 6 bei Ausführung der `switch`-Anweisung hat, werden nur die Wochentage Saturday und Sunday sowie der Text „there are no more weekdays“ am Bildschirm ausgegeben. Falls `weekdays` bei Eintritt in die Fallunterscheidung einen Wert besitzt, der nicht im Intervall  $[1,7]$  liegt, erscheint dagegen nur die Nachricht „there are no more weekdays“ auf dem Bildschirm.*

□



## 14 Wiederholungs- und Sprunganweisungen

Wiederholungsanweisungen, auch Schleifenanweisungen (engl. loop statement) genannt, ermöglichen es uns, eine Folge von Anweisungen in einem Programm wiederholt zu durchlaufen. Strukturierte Sprunganweisungen erlauben es, Schleifendurchläufe vorzeitig zu beenden oder die Schleife zu verlassen.

Wiederholungs-  
anweisung  
Schleifenanweisung

### 14.1 Bedingte Schleifen: `while` und `do ... while`

Eine bedingte Schleife wiederholt eine Anweisung, bei der es sich möglicherweise um einen Block handelt, solange eine boolesche Bedingung gültig ist.

#### Definition 14.1-1: `while`-Anweisung

Eine `while`-Schleife hat die Form:

```
while (expression)
    statement
```

`while`-Anweisung  
`while`-Schleife

wobei

- *expression* ein Ausdruck vom Typ `boolean` sein muss und
- *statement* eine Anweisung ist.

Während der Programmausführung wird eine `while`-Anweisung wie folgt ausgewertet:

1. Der Ausdruck *expression* wird ausgewertet. Falls das Ergebnis dieser Auswertung den Wert `true` liefert, wird die Anweisung *statement* ausgeführt, dann wird Schritt 1 wiederholt.
2. Falls die Auswertung von *expression* das Ergebnis `false` liefert, wird die Ausführung der Schleife beendet. [JLS: § 14.12]

Diese Semantik ist in Abbildung Abb. 14.1-1 dargestellt.

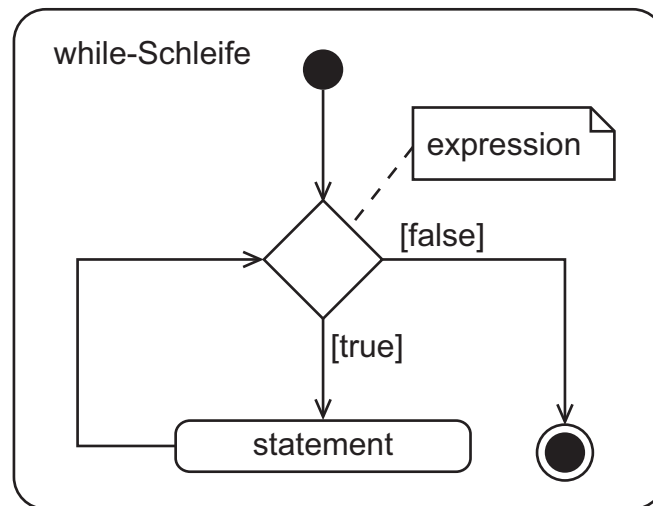
□

Eine `while`-Schleife wird demnach so lange ausgeführt, bis der Bedingungsdruck zu `false` evaluiert.

#### Beispiel 14.1-1: `while`-Schleife

Durch die Anweisungsfolge

```
int counter = 0;
while (counter < 10) {
    System.out.println("Java rocks!");
    counter++;
}
```



**Abb. 14.1-1:** Semantik einer while-Schleife

wird zehnmal die Zeichenkette "Java rocks!" geschrieben. Die Inkrementierung der Variablen counter in der letzten Zeile gewährleistet, dass nach dem zehnten Durchlauf der Bedingungsdruck zu false evaluiert und die while-Schleife verlassen werden kann.

□

#### **Bemerkung 14.1-1: Schleifenrumpf**

Schleifenrumpf Wir nennen das Innere einer Schleife, im Falle der while-Schleife aus Definition 14.1-1 die Anweisung statement, auch Schleifenrumpf.

□

☞ Für while-Schleifen wie für alle weiteren im Verlauf dieses Kapitels vorgestellten Schleifen empfehlen wir ihnen, stets Klammern { } zu verwenden, selbst wenn der Block nur eine Anweisung enthält. Die Platzierung der Klammern zeigt folgendes Beispiel.

```

while (expression) {
    statement1
    statement2
    ...
    statementn
}

```

☞ Es gilt die 4-Leerzeichen-Regel für Einrückungen. Falls eine expression getrennt werden muss, sollten Sie 8 Leerzeichen verwenden, da die 4-Leerzeichen-Einrückung die Erkennung des Anweisungsrumphs erschwert:

```

while ((condition1 && condition2)
      || (condition3 && condition4)) {
    statement;
}

```

**Selbsttestaufgabe 14.1-1:**

Zeichnen Sie ein Aktivitätsdiagramm zu Beispiel 14.1-1. Was passiert, wenn Sie `counter` mit 10 initialisieren?

**Selbsttestaufgabe 14.1-2:**

Schreiben Sie eine Folge von Anweisungen, die  $p^n$  für zwei `int`-Variablen `p` und `n` berechnet und weisen Sie das Ergebnis der Variablen `res` zu. Wählen Sie für `res` einen geeigneten Datentyp. Ihr Programm sollte alle notwendigen Deklarationen beinhalten. Die Wertzuweisung an `p` und `n` sollte offen bleiben.

**Selbsttestaufgabe 14.1-3:**

Schreiben Sie ein Programm, das ein ausgefülltes Rechteck aus `*` ausgibt. Das Rechteck soll aus `hoehe` Zeilen und `breite` Sternen in einer Zeile bestehen. Die Variablen `hoehe` und `breite` sind beide vom Typ `int`.

Sie können davon ausgehen, dass nur positive Werte für die beiden Variablen verwendet werden. Bei `breite = 5` und `hoehe = 4` soll folgende Ausgabe erscheinen:

```
*****
*****
*****
*****
```

Eine zweite Form der bedingten Schleife ist die `do`-Schleife.

**Definition 14.1-2: do-Schleife**

Die `do`-Schleife (engl. *do loop*) hat die folgende Syntax:

do-Schleife

```
do
    statement
while (expression);
```

Bei dieser Schleifenart wird während der Laufzeit zuerst die Anweisung `statement` ausgeführt. Dann wird der Bedingungsausdruck `expression` überprüft. Falls das Ergebnis `true` liefert, wird die Schleife so lange wiederholt, bis `expression` das Ergebnis `false` liefert. [JLS: § 14.13] Dieser Sachverhalt ist in Abb. 14.1-3 veranschaulicht.

□

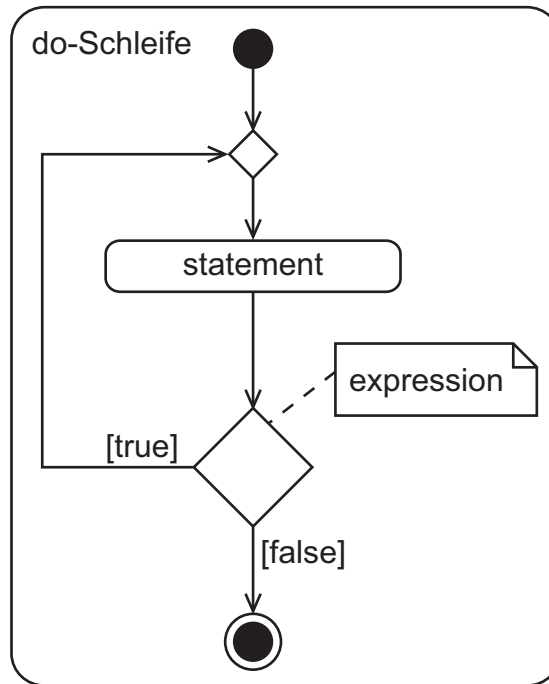


Abb. 14.1-3: Semantik einer do-Schleife

#### Bemerkung 14.1-2: Unterschied zwischen while- und do-Schleife

Im Gegensatz zur while-Schleife, bei der die Schleifenanweisung unter Umständen nie ausgeführt wird, wird bei einer do-Schleife die Schleifenanweisung immer mindestens einmal ausgeführt.

□

#### Bemerkung 14.1-3: Abweisende und nicht abweisende Schleifen

Schleifen, bei denen der Rumpf möglicherweise nie ausgeführt wird, nennen wir abweisende Schleifen. Wird der Rumpf immer mindestens einmal ausgeführt, so sprechen wir von nicht abweisenden Schleifen.

abweisende Schleife  
nicht abweisende  
Schleife

□

#### Selbsttestaufgabe 14.1-4:

Ändern Sie Beispiel 14.1-1 in eine do-Schleife um und zeichnen Sie ein Aktivitätsdiagramm. Was passiert, wenn Sie counter mit 10 initialisieren?

## 14.2 Die Zähl- oder for-Schleife

for-Schleife

Mit einer for-Schleife können wir eine Schleife durch eine Zählvariable steuern. Ausgehend von einem Anfangswert läuft die Schleifenvariable bei einer definierten Schrittweite bis zu einem Endwert, und bei jedem Schritt wird die Schleifenanweisung ausgeführt. Über den Anfangs- und Endwert sowie die Schrittweite wird

bei der `for`-Schleife die Anzahl der Wiederholungen von der Programmiererin bestimmt.

### Definition 14.2-1: `for`-Schleife

Eine `for`-Schleife hat folgende Form:

```
for (forInit; booleanExpr; forUpdate)
    statement
```

wobei

- der optionale Ausdruck `forInit` im Normalfall eine Liste aus einer oder mehreren Variablendeklarationen und Initialbelegungen enthält, die durch Kommata voneinander getrennt werden; die neu eingeführten Schleifenvariablen müssen alle vom gleichen Typ sein; sie sind außerhalb der `for`-Schleife nicht sichtbar; der Ausdruck `forInit` wird genau einmal zu Beginn der Ausführung der Schleife evaluiert;
- `booleanExpr` ein optionaler Vergleichsausdruck vom Typ `boolean` ist; er bestimmt den Endwert und wird vor jedem Wiederholungszyklus der Schleife überprüft; sobald die Auswertung `false` liefert, wird die Schleife beendet;
- `forUpdate` eine optionale, durch Kommata getrennte Folge von einer oder mehreren Anweisungen ist, die im Normalfall die Schleifenvariablen verändern; der Ausdruck wird in jedem Schleifendurchlauf nach der Ausführung von `statement` ausgewertet;
- `statement` eine Anweisung ist. [JLS: § 14.14.1]

Dieser Sachverhalt ist in Abb. 14.2-1 veranschaulicht.

□

### Beispiel 14.2-1: `for`-Schleife

Das folgende Beispiel:

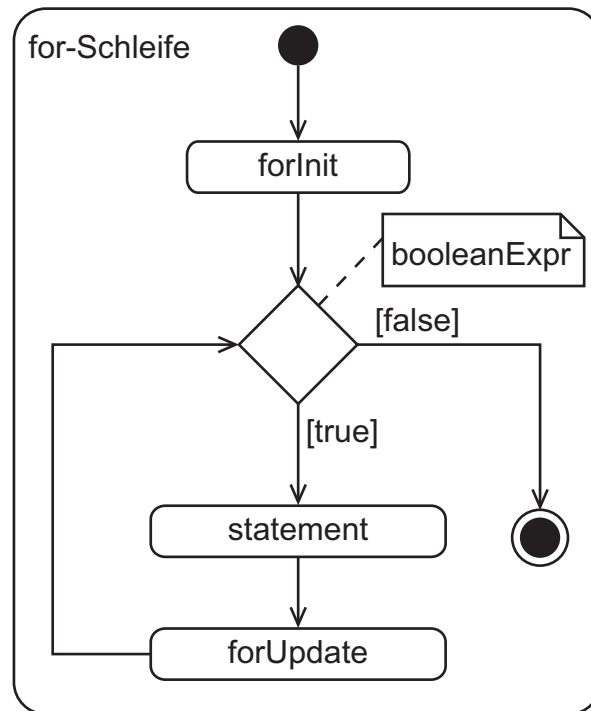
```
for (int i = 0; i < 5 ; i++) {
    System.out.println(i);
}
```

wird auf Ihrem Computerbildschirm folgendes ausgegeben:

```
0
1
2
3
4
```

Das Programmstück:

```
for (char c = 'f'; c >= 'a'; c--) {
    System.out.println(c);
}
```



**Abb. 14.2-1:** Semantik einer `for`-Schleife

erzeugt auf dem Bildschirm folgende Ausgabe:

f  
e  
d  
c  
b  
a

□

☞ Bei der Vergabe von Namen für Schleifenvariablen darf von der Regel, Namen beschreibend und selbsterklärend zu wählen, abgewichen werden. Schleifenvariablen werden oft durch einzelne Buchstaben dargestellt, weil ihre Funktion meist aus dem Kontext heraus klar ist.

**Bemerkung 14.2-1:**

*Ist keine `booleanExpr` vorhanden, so verhält sich die Schleife genauso, als wäre der Ausdruck zu `true` evaluiert worden.*

□

In einer späteren Lerneinheit werden wir eine zweite Form von Zählschleifen kennen lernen, die im Zusammenhang mit Datensammlungen und Feldern zum Einsatz kommt.

**Selbsttestaufgabe 14.2-1:**

Handelt es sich bei der `for`-Schleife um eine abweisende Schleife oder nicht? Begründen Sie Ihre Antwort.

**Selbsttestaufgabe 14.2-2:**

Transformieren Sie das Schema für den Aufbau von Zählschleifen in Definition 14.2-1 in eine semantisch gleichwertige `while`-Schleife.

**Selbsttestaufgabe 14.2-3:**

Gegeben sei eine initialisierte `int`-Variable `testzahl`. Konstruieren Sie eine `for`-Schleife, die aufsteigend alle Teiler von `testzahl` ausgibt.

**Lösungshinweis:** Zum Testen Ihrer Lösung können Sie die Initialisierung (mit einem beliebigen Wert) der `for`-Schleife voranstellen:

```
int testzahl = 100;
for ...
...
```

Eine Zahl  $a$  ist ein Teiler einer Zahl  $b$ , wenn der Rest der Division  $b / a$  den Wert 0 hat.

**Selbsttestaufgabe 14.2-4:**

Wie oft wird der Rumpf der folgenden `for`-Schleifen ausgeführt, wenn der Rumpf die Variable `i` nicht verändert?

a)

```
for (int i = -10; i > 10; i++) { ... }
```

b)

```
for (int i = -10; i <= 10; i++) { ... }
```

c)

```
for (int i = 2; i >= -3; i--) { ... }
```

d)

```
for (int i = -20; i <= 5; i += 4) { ... }
```

e)

```
for (int i = 7; i >= -10; i = i - 3) { ... }
```

## 14.3 Strukturierte Sprunganweisungen: `break` und `continue`

strukturierte  
Sprunganweisung

Im Rumpf von Schleifenanweisungen können die strukturierten Sprunganweisungen `break` und `continue` benutzt werden, um den Fluss von Wiederholungsanweisungen zu unterbrechen.

### Definition 14.3-1: `break`

break-Anweisung

*In allen Schleifen bewirkt die Ausführung einer `break`-Anweisung ohne Sprungmarke, dass die innerste Schleife, in der die `break`-Anweisung auftritt, sofort verlassen wird, ohne die restlichen Anweisungen im jeweiligen Block auszuführen. Die Programmausführung wird nach dem betroffenen Schleifenblock fortgesetzt.*

*Die `break`-Anweisung ohne Sprungziel kann auch in Mehrfach-Fallunterscheidungen verwendet werden, um die Ausführung der auf den ausgewählten Fall folgenden Anweisungen zu unterbinden. [JLS: § 14.15]*

□

### Beispiel 14.3-1: `break`-Anweisungen in einer Mehrfach-Fallunterscheidung

*Das folgende Programmstück illustriert den Gebrauch der `break`-Anweisung in einer `switch`-Anweisung:*

```
int weekdays;
...
switch (weekdays) {

case 1:
    System.out.println("Monday");
    break;
case 2:
    System.out.println("Tuesday");
    break;
case 3:
    System.out.println("Wednesday");
    break;
case 4:
    System.out.println("Thursday");
    break;
case 5:
    System.out.println("Friday");
    break;
case 6:
    System.out.println("Saturday");
    break;
case 7:
    System.out.println("Sunday");
    break;
default:
    System.out.println("there are no more weekdays");
}
```



Welcher Fall aus 1 bis 7 auch immer bei der Ausführung dieser Fallunterscheidung zutrifft: das Programm wird immer nur den zugehörigen Wochentag auf Ihrem Bildschirm ausgegeben, da die `break`-Anweisungen verhindern, dass der Kontrollfluss über die nachfolgenden Anweisungen fließt.

□

### Beispiel 14.3-2: `break`-Anweisung in einer `do`-Schleife

Gegeben seien die folgenden Anweisungen:

```
int x = 317;
do {
    if (x % 17 == 0) {
        break;
    }
    x++;
} while (true);
System.out.println(x);
```

Bei der Ausführung erscheint die Ausgabe

323

auf dem Bildschirm, da die Schleife auf Grund der `break`-Anweisung genau dann verlassen wird, wenn  $x$  durch 17 teilbar ist. Der Kontrollfluss ist in Abb. 14.3-1 dargestellt.

□

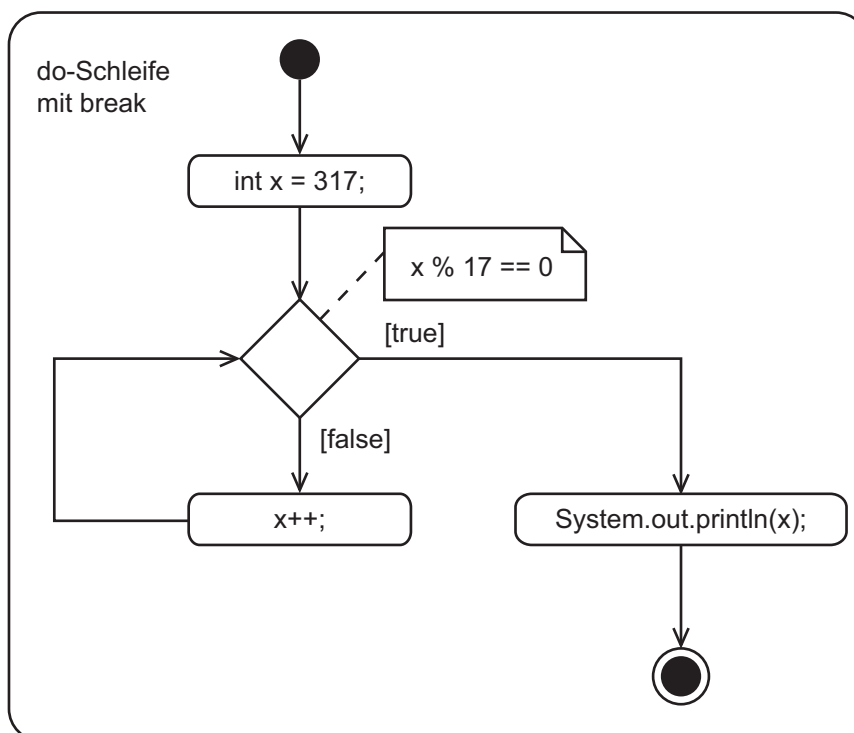


Abb. 14.3-1: `break`-Anweisung in einer `do`-Schleife

**Selbsttestaufgabe 14.3-1:**

Entwickeln Sie ein Programm, das abhängig von dem Wert der Variablen `monat` vom Typ `byte` ausgibt, wie viele Tage dieser Monat hat. Sollte `monat < 1` oder `monat > 12` sein, soll Es gibt keinen solchen Monat ausgegeben werden.

Bei `monat = 5` soll zum Beispiel 31 Tage ausgegeben werden. Bei Februar soll immer 28/29 Tage ausgegeben werden.

**Definition 14.3-2: continue-Anweisung**

`continue`-Anweisung

Die Anweisung `continue` stoppt die Ausführung der Wiederholungsanweisung, in der sie vorkommt, springt zum Anfang der Schleifenanweisung und startet unverzüglich den nächsten Schleifendurchlauf. [JLS: § 14.16]

□

**Beispiel 14.3-3: continue in einer for-Schleife**

Gegeben seien die folgenden Anweisungen:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        i = i + 2;
        continue;
    }
    System.out.println(i);
}
```

Bei der Ausführung erscheint die folgende Ausgabe auf dem Bildschirm:

```
0
1
2
3
4
8
9
```

Wenn `i` den Wert 5 annimmt, wird `i` zuerst um 2 erhöht und anschließend direkt die Anweisung `forUpdate (i++)` ausgeführt. Somit ist in `i` der Wert 8 gespeichert und die Schleife wird fortgesetzt. Der Kontrollfluss ist in Abb. 14.3-2 veranschaulicht.

□

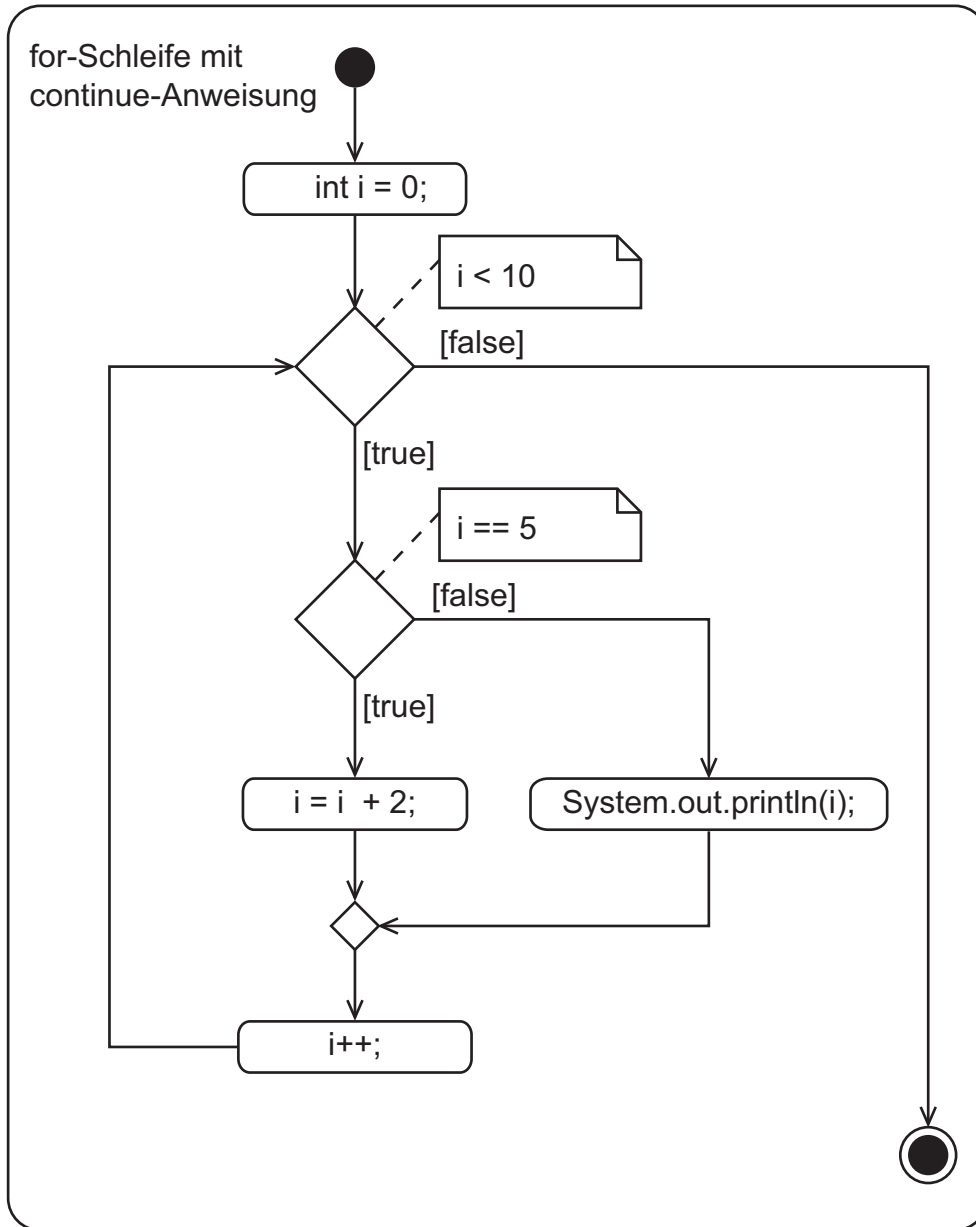


Abb. 14.3-2: continue-Anweisung in einer for-Schleife

**Beispiel 14.3-4: Schleife mit break- und continue-Anweisung**

*Eine while-Schleife mit break- und continue-Anweisung ist in der unten stehenden Anweisungssequenz dargestellt:*

```
int summe = 0;
while (summe < 100) {
    summe += 1;
    if (summe == 20) {
        System.out.println("Schluss!");
        break;
    }
    if (summe < 16) {
        summe += 3;
        continue;
    }
    System.out.println(summe);
}
```

□

**Selbsttestaufgabe 14.3-2:**

*Erstellen Sie ein Aktivitätsdiagramm zu dem Programmstück aus Beispiel 14.3-4. Welche Folge von Ausgaben produziert es?*

## 15 Gültigkeitsbereich und Sichtbarkeit von lokalen Variablen

Variablen werden durch ihre Vereinbarung in einem Programm eingeführt. Auf sie kann in allen Anweisungen, die innerhalb des Blocks und nach der Variablenvereinbarung stehen, zugegriffen werden. Die Variablen sind auch in inneren Blöcken sichtbar. Wir machten uns diese Regel bereits mehrfach in verschachtelten Schleifenanweisungen, die auch Blöcke sind, zu Nutze.

Der Gültigkeitsbereich (engl. scope) einer Variablendeklaration ist der Rest des Blocks, inklusive aller inneren Blöcke, in dem sie sich befindet. [JLS: § 14.4.2] Gültigkeitsbereich

### Bemerkung 15-1: Sichtbarkeitsbereich

*Jeder Block, inklusive all seiner enthaltenen Blöcke, ist ein eigener Sichtbarkeitsbereich.* Sichtbarkeitsbereich

□

### Bemerkung 15-2: Wiederverwendung von Namen

*Innerhalb eines Sichtbarkeitsbereichs können Variablennamen nicht erneut in Deklarationen benutzt werden.*

*Deklarationen mit identischen Variablennamen in getrennten Sichtbarkeitsbereichen sind zulässig.*

□

---

#### Programm 15-1: Gültigkeitsbereiche und verschachtelte Blöcke

---

```

{ // M: äußerster Block
    // sichtbar sind:
    final int MAX = 7; // MAX ab hier
    int summe = 0; // summe ab hier
    boolean gefunden = false; // gefunden ab hier

    for (int i = 0; i < MAX; i++) { // S1: innerer Block zu M
        // sichtbar sind:
        // MAX, summe, gefunden
        // i ab hier
        summe += i;
    } // i bis hier
    // Block S1 beendet

    for (int j = 0; j < MAX; j++) { // S2: innerer Block zu M
        // sichtbar sind:
        // MAX, summe, gefunden
        // j ab hier
        System.out.println(j);
    } // j bis hier
    // Block S2 beendet

    for (int i = MAX - 1; i >= 0; i--) { // S3: innerer Block zu M
        // sichtbar sind:

```

```

//      MAX, summe, gefunden
//      i ab hier
int treffer = 10; //      treffer ab hier
if (!gefunden) { // F1: innerer Block zu S3
//      sichtbar sind:
//      MAX, summe, gefunden
//      i, treffer
    int temp = summe; //      temp ab hier
    temp -= i;
    if (temp == treffer) { // F2: innerer Block zu F1
//      sichtbar sind:
//      MAX, summe, gefunden
//      i, treffer, temp
        gefunden = true;
    } //      Block F2 beendet
} //      temp bis hier
//      Block F1 beendet
} //      i, treffer bis hier
//      Block S3 beendet

System.out.println(gefunden);
} //      MAX, summe, gefunden
//      bis hier
//      Block M beendet

```

Abb. 15-1 veranschaulicht das Konzept des Gültigkeitsbereiches und der Sichtbarkeit von Variablennamen für das Beispiel in Programm 15-1. Obgleich kein guter Programmierstil, darf in Programm 15-1 der Name `i` in den Blöcken `S1` und `S3` wiederverwendet werden, weil sich die Blöcke nicht überschneiden. Hätten wir aber anstelle von `temp` den Namen `i`, `treffer`, `summe` oder `gefunden` in Block `F2` gewählt, wäre das Programm illegal geworden.

### Selbsttestaufgabe 15-1:

*Welche der folgenden Deklarationen ist ungültig und warum?*

```

int x = 10; // 1
double i = 10, z; // 2
int k = 5; // 3
int y = (int) z; // 4
float x = 3.4f, u; // 5
for (int k = 0, v; k < 10; k++) { // 6
    int w = 5; // 7
    k = w * 2; // 8
} // 9
double w = 2; // 10
boolean v = true; // 11

```

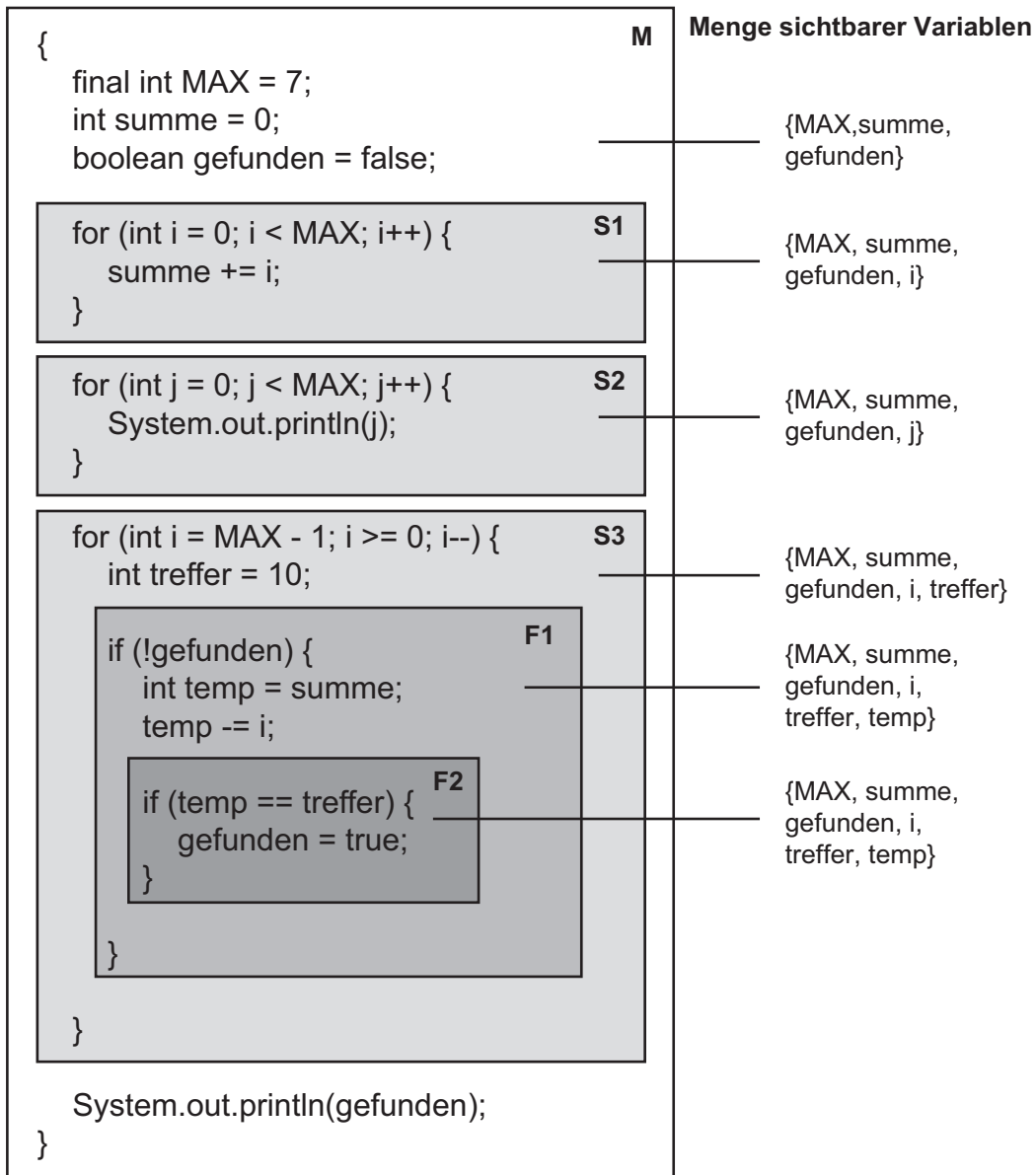


Abb. 15-1: Sichtbarkeit von Namen

## 16 Zusammenfassung

In dieser Kurseinheit haben wir primitive Datentypen, Deklarationen, Ausdrücke, Variablen, Zuweisungen und Anweisungen kennen gelernt, um den Programmfluss bestimmen zu können. Im Einzelnen wissen wir:

**Primitive Datentypen** ermöglichen die Darstellung verschiedener Werte, wie zum Beispiel **Zahlen**, **Zeichen** und **Wahrheitswerte**. Zu jedem Datentyp gibt es eine Menge an definierten **Operatoren** um die Werte zu verknüpfen.

**Literale** sind Folgen von Symbolen, deren Wert vom Computer „buchstäblich verstanden“ wird. Literale erlauben uns, Werte einfachen Typs als Teil eines Programmtextes niederzuschreiben.

Eine **Variable** speichert einen veränderbaren Wert. Mit einer **Variablendeklaration** werden ihr Name und Typ festgelegt. Sowohl der Name als auch der Typ können nicht verändert werden. Die Variable ist nur in ihrem **Gültigkeitsbereich** sichtbar.

Eine Variable kann bei ihrer Vereinbarung initialisiert, d. h. auf einen Anfangswert gesetzt werden. Der Wert einer Variablen wird durch **Zuweisungen** verändert.

Eine **final-Variable** kann genau einmal, entweder bei ihrer Deklaration oder zu einem späteren Zeitpunkt, mit einem Wert initialisiert werden. Dieser Wert ist dann final für diese Variable und kann nicht mehr verändert werden.

Ein **Ausdruck** ist eine Folge von Variablen, Operatoren, Klammern und Literalen, die zu einem einzelnen Wert ausgewertet werden kann.

Der Datentyp eines Ausdrucks hängt von den Elementen ab, die in diesem Ausdruck benutzt wurden.

Ausdrücke werden in der Reihenfolge des **Vorrangs der Operatoren**, die im jeweiligen Ausdruck vorkommen, berechnet.

Java führt verschiedene **Typumwandlungen** automatisch durch, wenn Ausdrücke und Variablen von unterschiedlichen, aber kompatiblen Typen in Ausdrücken, Zuweisungen oder Anweisungen verwendet werden. Wir nennen dies eine **implizite Typumwandlung**.

In Zuweisungen legt der Typ der Variablen den Typ des Zuweisungsausdrucks fest. Infolgedessen wird es, wenn der rechtsseitige Ausdruck der Zuweisung einen Typ beinhaltet, der größer ist als der Typ der Variablen auf der linken Seite, zu einem Übersetzungsfehler kommen, da Typen in Java nicht automatisch eingeeignet werden. Eine Typumwandlung kann mit Hilfe des **cast-Operators** erzwungen werden. Wir sprechen dann von einer **expliziten Typumwandlung**.

Im Unterschied zu Ausdrücken und Zuweisungen haben **Blöcke** und **Kontrollstrukturen** weder einen Wert noch einen Typ. Sie besitzen wünschenswerte Effekte auf veränderliche Eigenschaften eines Programms und auf die genutzte Umgebung.



---

Eine **if-Anweisung** erlaubt die alternative Ausführung von Anweisungen oder Anweisungsblöcken in Abhängigkeit von einer booleschen Bedingung.

Eine **switch-Anweisung** kann, abhängig vom Wert eines Ausdrucks, zwischen einer Anzahl alternativer Anweisungen auswählen. Sie wirkt wie ein Mehrwegschalter.

**Wiederholungsanweisungen** erlauben es uns, Folgen von Anweisungen zu wiederholen. Wenn eine Aufgabe so gelöst werden muss, dass die Schleife durch eine Bedingung verlassen wird und die Iteration mindestens einmal durchgeführt werden muss, wird die **do-Schleife** eingesetzt. Andernfalls verwendet man die **while-Schleife**. Die **for-Schleife** wird dann benötigt, wenn die Schleife (häufig über einen ganzzahligen Zähler) in einer vorgegebenen Schrittweite und -zahl laufen muss. Zur Modifikation des Kontrollflusses haben wir die **strukturierten Sprunganweisungen break** und **continue** kennen gelernt.

Nachdem wir verschiedene Anweisungsarten der Sprache Java kennen lernten, sind wir in der Lage, komplexe Kontrollstrukturen zu formulieren. Neben den elementaren Datentypen, die nicht viel Gestaltungsspielraum lassen, fehlen uns jedoch noch geeignete Konzepte, um sinnvolle Anwendungen programmieren zu können. Solche Sprachkonzepte sind Gegenstand der nächsten Kurseinheit.



---

## Lösungen zu Selbsttestaufgaben der Kurseinheit

### *Lösung zu Selbsttestaufgabe 9.1-1:*

Die Aussage ist falsch. Die Ziffern des oktalen Wertebereichs liegen zwischen 0 und 7.

### *Lösung zu Selbsttestaufgabe 9.1-2:*

23L ist ein Wert vom Typ `long`,  
7 ist vom Typ `byte`,  
128 ist vom Typ `int`,  
.4 ist vom Typ `double`,  
07 ist eine Oktalzahl vom Typ `byte`.

### *Lösung zu Selbsttestaufgabe 9.1-3:*

697 ist die korrekte Antwort.

### *Lösung zu Selbsttestaufgabe 9.1-4:*

Da der Wert 3456789012 außerhalb des Wertebereichs des Datentyps `int` liegt, handelt es sich um ein ungültiges Literal. Es ist aber möglich, durch Anhängen des Zeichens `L` ein gültiges Literal vom Typ `long` zu erzeugen: 3456789012L.

### *Lösung zu Selbsttestaufgabe 9.2-1:*

Das erste und zweite Literal sind vom Typ `double`, das dritte vom Typ `float`, und die letzte Zeichenfolge ist kein zulässiges Literal in Java.

**Lösung zu Selbsttestaufgabe 9.2-2:**

- 1234 ist vom Typ `int`
- 2f ist vom Typ `float`
- 44.45 ist vom Typ `double`
- 2e1f ist vom Typ `float`
- 0x2e1f ist vom Typ `int`
- .3 ist vom Typ `double`
- 2124L ist vom Typ `long`
- 41D ist vom Typ `double`
- 6d ist vom Typ `double`
- 0x88 ist vom Typ `int`
- 3E10 ist vom Typ `double`

**Lösung zu Selbsttestaufgabe 9.3-1:**

Der Ausdruck ist syntaktisch wohlgeformt und liefert das Ergebnis 6. Hinweise zur Auswertungsreihenfolge finden Sie im nächsten Abschnitt.

**Lösung zu Selbsttestaufgabe 9.4-1:**

Die Auswertung ergibt:

21  
4  
5

**Lösung zu Selbsttestaufgabe 9.4-2:**

Der Ausdruck  $((12 * 5) \% (-3)) - (5 / 3) + 7$  aus Zeile 1 ist äquivalent zum Ausdruck  $12 * 5 \% -3 - 5 / 3 + 7$ .

Der Ausdruck  $(12 * (5 \% (-3))) - (5 / 3) + 7$  in Zeile 2 ist nicht äquivalent. Er liefert den Wert 30.

Der Ausdruck  $((12 * (5 \% (-3 - 5))) / 3) + 7$  in Zeile 3 ist nicht äquivalent. Er liefert den Wert 27.

Auch der Ausdruck  $(12 * (5 \% (-3)) - (5 / 3)) + 7$  in Zeile 4 ist nicht äquivalent. Er liefert den Wert 30.

**Lösung zu Selbsttestaufgabe 9.5-1:**

Die Auswertung ergibt:

```
false
true
false
true
true
true
false
```

**Lösung zu Selbsttestaufgabe 9.6-1:**

Die Vergleiche liefern:

```
false
false
true
false
true
```

**Lösung zu Selbsttestaufgabe 10.1-1:**

In der Variablendeklaration `int x y z;` fehlt ein Komma nach `x` und `y`.

In der Deklaration `double currency Rate;` fehlt entweder ein Komma nach `currency`, falls zwei Variablen vereinbart werden sollten, oder der beabsichtigte Variablenname muss in einem Wort (ohne Leerzeichen) geschrieben werden.

Die Deklaration `int null;` ist unzulässig, weil `null` ein reserviertes Wort der Sprache Java ist.

Die beiden Deklarationen `int x, int y;` müssen durch ein Semikolon statt des Kommas getrennt werden.

Die Deklaration `bool b` enthält zwei Fehler: das Schlüsselwort heißt `boolean`, und die Vereinbarung muss mit einem Semikolon abgeschlossen werden.

Die Deklaration `int 3times;` ist unzulässig, weil der Bezeichner mit einem Buchstaben beginnen muss.

**Lösung zu Selbsttestaufgabe 10.1-2:**

Die Bezeichner `3D`, `4you` und `this` sind unzulässig, weil Bezeichner nicht mit einer Ziffer beginnen dürfen und `this` ein reserviertes Wort der Sprache ist. Zudem entspricht der Name `HEIGHT` nicht der Konvention, dass Variablennamen mit Kleinbuchstaben beginnen sollen. Diese Schreibweise nutzen wir nur bei finalen Variablen (siehe Abschnitt 10.2). Die Variablennamen `this_position` und `any_Float` sollten auch lieber ohne Unterstrich geschrieben werden: `thisPosition` und `anyFloat`.

**Lösung zu Selbsttestaufgabe 10.2-1:**

```
double area = height * base / 2.0;
```

**Lösung zu Selbsttestaufgabe 10.2-2:**

Nach der Ausführung der drei Zuweisungen in den Zeilen 1-3 haben die Variablen `x`, `y` und `z` die Werte 5, 4 und 3. Durch die Anweisung in Zeile 4 wird `x` auf den Wert 4 gesetzt, die anderen Variablen tragen noch ihren Initialwert. Nach Ausführung der Anweisung in Zeile 5 liegt die Belegung weiterhin bei 4, 4, 3, da `y` erneut der Wert 4 zugewiesen wurde. Nach Ausführung der letzten Zeile ergibt sich die Belegung 4, 4, 11.

**Lösung zu Selbsttestaufgabe 10.2-3:**

a)

```
double g = ...;
double t = ...;
double s;
s = 0.5 * g * t * t;
```

b)

```
final double PI = 3.14;
double r = ...;
double a;
a = PI * r * r;
```

c)

```
final double PI = 3.14;
double h = ...;
double r1 = ...;
double r2 = ...;
double v;
v = (PI * h) / 3 * (r1 * r1 + r1 * r2 + r2 * r2);
```

**Lösung zu Selbsttestaufgabe 10.2-4:**

Anweisung c) ist die richtige Antwort.

Hier noch einmal die drei Anweisungen mit den jeweiligen Resultaten für x und n:

a)

```
int n = 0;
int x = 1;
n = x++ + x++;    // n == 3 und x == 3
```

b)

```
int n = 0;
int x = 1;
n = n++ - x++;    // n == -1 und x == 2
```

c)

```
int n = 0;
int x = 1;
n = x-- + -x++;    // n == 1 und x == 1
```

**Lösung zu Selbsttestaufgabe 10.2-5:**

```
(x < 10) || (x % 2 == 0 && x > 20)
```

Mit Hilfe von `x % 2 == 0` prüft man, ob es sich bei x um eine gerade Zahl handelt.

**Lösung zu Selbsttestaufgabe 10.2-6:**

```
y % x == 0
```

Wenn x ein Teiler von y ist, so muss die Division  $y / x$  den Rest 0 ergeben.

**Lösung zu Selbsttestaufgabe 11.1-1:**

```
char buchstabel = ...;
char buchstabe2 = ...;
final int ABSTAND = 'a' - 'A';
boolean sindGleich = (buchstabel == buchstabe2)
                    || (buchstabel == buchstabe2 - ABSTAND)
                    || (buchstabe2 == buchstabel - ABSTAND);
```

**Lösung zu Selbsttestaufgabe 11.1-2:**

Die Ausdrücke `129` und `12.5` können wegen des möglichen Verlusts an Präzision der Zahlendarstellung nicht an eine `byte`-Variable zugewiesen werden.

Die anderen Ausdrücke sind als Operanden einer Zuweisung an `b` zulässig und liefern folgende Werte: `6`, `102` und `78`.

**Lösung zu Selbsttestaufgabe 11.1-3:**

`x` hat den Wert `9`, `y` den Wert `2.5` und `erg` den Wert `10.5`.

**Lösung zu Selbsttestaufgabe 11.2-1:**

Das Problem ist, dass es sich bei dem Ausdruck `prozentsatz / 100` um eine ganzzahlige Division handelt, die zum Wert `0` evaluiert. Der Fehler kann behoben werden, indem zum Beispiel die Reihenfolge vertauscht wird.

```
long jahreszinsInCent = betragInCent * prozentsatz / 100;
```

Diese Lösung vermeidet auch unnötige explizite Typumwandlungen. Will man die Reihenfolge nicht ändern, so benötigt man eine explizite Typumwandlung:

```
long jahreszinsInCent =
    (long) (prozentsatz / 100.0 * betragInCent);
```

**Lösung zu Selbsttestaufgabe 11.2-2:**

Es müssen mindestens zwei Typumwandlungen stattfinden. Zunächst ist eine `int`-Division zu verhindern, wofür uns verschiedene implizite und explizite Erweiterungen als Lösung zur Auswahl stehen. Diese bewirken eine Gleitkomma-Division und ein Zwischenergebnis in einem Gleitkommatyp, das vor der Zuweisung explizit auf den Typ `int` eingeeengt werden muss. Einige mögliche korrekte Zuweisungen sind:

```
celsius = (int) (5.0 / 9 * (fahrenheit - 32));
celsius = (int) (5 / 9.0 * (fahrenheit - 32));
celsius = (int) ((double) 5 / 9 * (fahrenheit - 32));
celsius = (int) ( 5 / (double) 9 * (fahrenheit - 32));
```



**Lösung zu Selbsttestaufgabe 11.2-3:**

Alle drei Initialisierungen werden vom Java-Übersetzer akzeptiert, und bei keiner entsteht ein Informationsverlust. Allerdings hat auch keine der durchgeführten Typumwandlungen einen Nutzen.

Der `int`-Wert 32 in der ersten Initialisierung ist klein genug, um verlustfrei auf den Typ `byte` eingeengt zu werden. Durch die Zuweisung an eine `int`-Variable wird er anschließend wieder auf den Typ `int` ausgeweitet.

In der zweiten Initialisierung wird der Wert 32 erneut explizit auf den Typ `byte` eingeengt, um sogleich vor der Multiplikation mit 123.45 implizit auf `double` ausgeweitet zu werden. Bei der Zuweisung des Ergebnisses an `preis` findet keine Umwandlung mehr statt.

Die dritte Initialisierung enthält eine explizite Umwandlung eines `boolean`-Wertes in den Typ `boolean`. Jeder Typ - auch der Typ `boolean`, der sich ansonsten jeder Umwandlung widersetzt - kann in sich selbst umgewandelt werden. Natürlich ist dies wirkungslos.

**Lösung zu Selbsttestaufgabe 11.2-4:**

```
double durchschnitt = ((double) a + b + c) / 3;
```

Alternativ könnte man auch die explizite Typumwandlung weglassen und stattdessen das `double`-Literal 3. verwenden. Jedoch würde dies bei großen Werten für `a`, `b` und `c` zu einem Überlauf führen, der so vermieden werden kann. Durch die `double`-Division können jedoch immer Ungenauigkeiten auftreten, zum Beispiel für `a = 3000000000000000000L`, `b = 2` und `c = 1`.

**Lösung zu Selbsttestaufgabe 13.1-1:**

```
double preis = ...;
if (preis > 100) {
    preis *= 0.97;
}
double endpreis = preis * 1.19;
```

**Lösung zu Selbsttestaufgabe 13.1-2:**

```

boolean istGrossbuchstabe;
char testZeichen = 'F';

if (testZeichen >= 'A' && testZeichen <= 'Z') {
    istGrossbuchstabe = true;
} else {
    istGrossbuchstabe = false;
}

System.out.println(istGrossbuchstabe);

```

Die Aufgabe lässt sich aber auch ohne if-Anweisung schreiben:

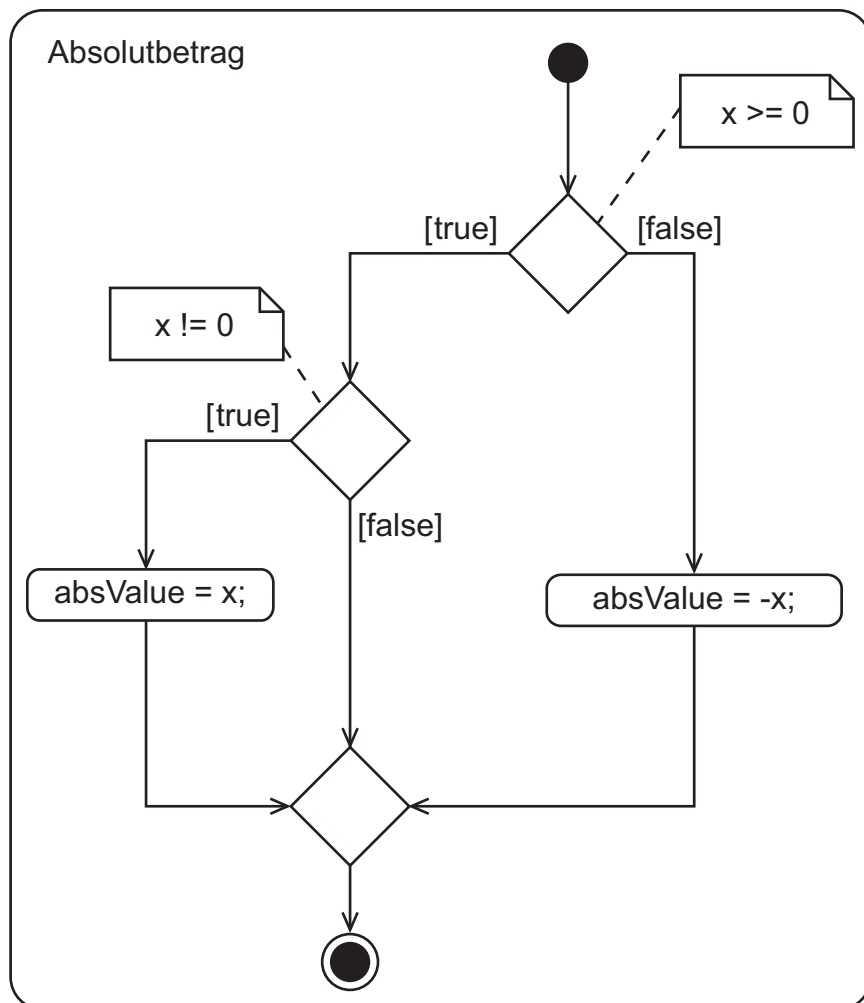
```

istGrossbuchstabe = (testZeichen >= 'A') && (testZeichen <= 'Z');

```

**Lösung zu Selbsttestaufgabe 13.1-3:**

Abb. 13.1-3 zeigt die verschachtelte if-Anweisung aus Beispiel 13.1-2.



**Abb. 13.1-3:** Verschachtelte if-Anweisung

**Lösung zu Selbsttestaufgabe 13.1-4:**

- a) Die Bedingung in einer `if`-Anweisung muss immer vom Typ `boolean` sein. Der Typ der Zuweisung `x = 1` ist jedoch `int`.
- b) Da der Operator `==` einen höheren Vorrang als `&&` hat, wird dieser zuerst ausgewertet und liefert einen Wert vom Typ `boolean` zurück. Der Operator `&&` ist jedoch nur auf zwei Werten vom Typ `boolean` definiert. Bei `x` handelt es sich jedoch um eine Variable vom Typ `int`, welche in Java nicht in `boolean` umgewandelt werden kann.
- c) Das Schlüsselwort `then` gibt es in Java nicht, und der Bedingungsausdruck einer `if`-Anweisung muss immer in runden Klammern stehen.

**Lösung zu Selbsttestaufgabe 13.1-5:**

```

int start = 800;
int ende = 1645;

// berechne die Stunden und Minuten
int stundeStart = start / 100;
int minStart = start % 100;
int stundeEnde = ende / 100;
int minEnde = ende % 100;

// umrechnen der Uhrzeiten in ganze Minuten
int startMinuten = stundeStart * 60 + minStart;
int endeMinuten = stundeEnde * 60 + minEnde;
int differenzMinuten;
if (endeMinuten >= startMinuten) {

    // beide Uhrzeiten beziehen sich auf den gleichen Tag
    differenzMinuten = endeMinuten - startMinuten;
} else {

    // beide Uhrzeiten beziehen sich auf verschiedene Tage
    // berechne Differenz zum Ende des ersten Tages
    differenzMinuten = 24 * 60 - startMinuten;

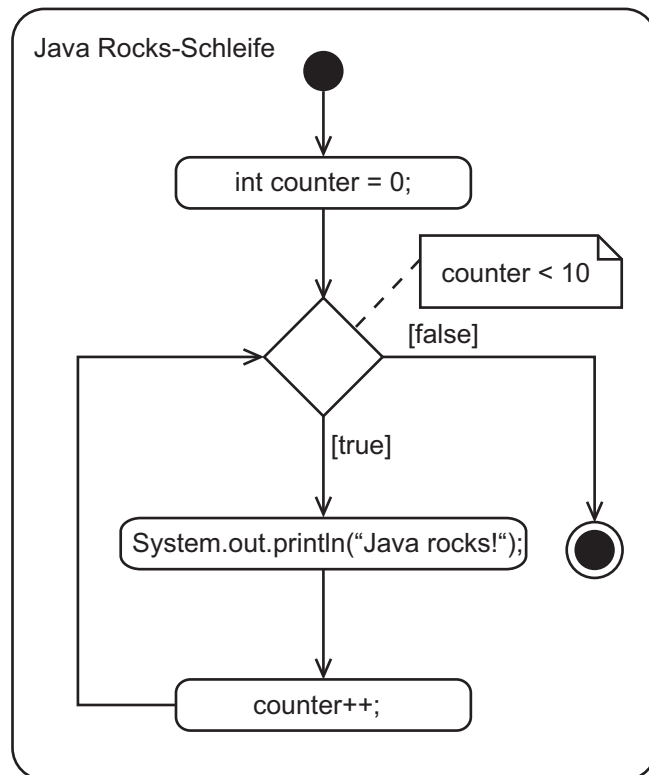
    // addiere die Minuten des 2. Tages
    differenzMinuten += endeMinuten;
}

// berechne Minuten und Stunden
int min = differenzMinuten % 60;
int stunde = differenzMinuten / 60;
System.out.print(stunde);
System.out.println(" Stunden");
System.out.print(min);
System.out.println(" Minuten");

```

**Lösung zu Selbsttestaufgabe 14.1-1:**

Abb. 14.1-2 zeigt die while-Schleife aus Beispiel 14.1-1.



**Abb. 14.1-2:** Aktivitätsdiagramm zu Beispiel 14.1-1

Wenn counter mit 10 initialisiert wird, evaluiert der Bedingungsausdruck sofort zu false, so dass die Schleife keinmal ausgeführt wird.

**Lösung zu Selbsttestaufgabe 14.1-2:**

```

int p, n;
... // Wertzuweisung an p und n
double res = 1; // bei negativen n reichen keine
// Ganzzahlen

if (n < 0) {
    if (p == 0) { // hier würde sonst eine Division
                // durch Null erfolgen
        System.out.println(
            "Dieses Ergebnis kann nicht berechnet werden!");
    } else {
        int i = 0;
        while (i > n) {
            res *= p;
            i--;
        }
        res = 1.0 / res; // Kehrwert bilden, da n negativ ist
    }
} else { // n >= 0

```

```
int i = 0;
while (i < n) {
    res *= p;
    i++;
}
}
```

### ***Lösung zu Selbsttestaufgabe 14.1-3:***

```
int breite = ...;
int hoehe = ...;
for (int h = 0; h < hoehe; h++) {

    // gebe breite Sterne aus
    for (int b = 0; b < breite; b++) {
        System.out.print("*");
    }

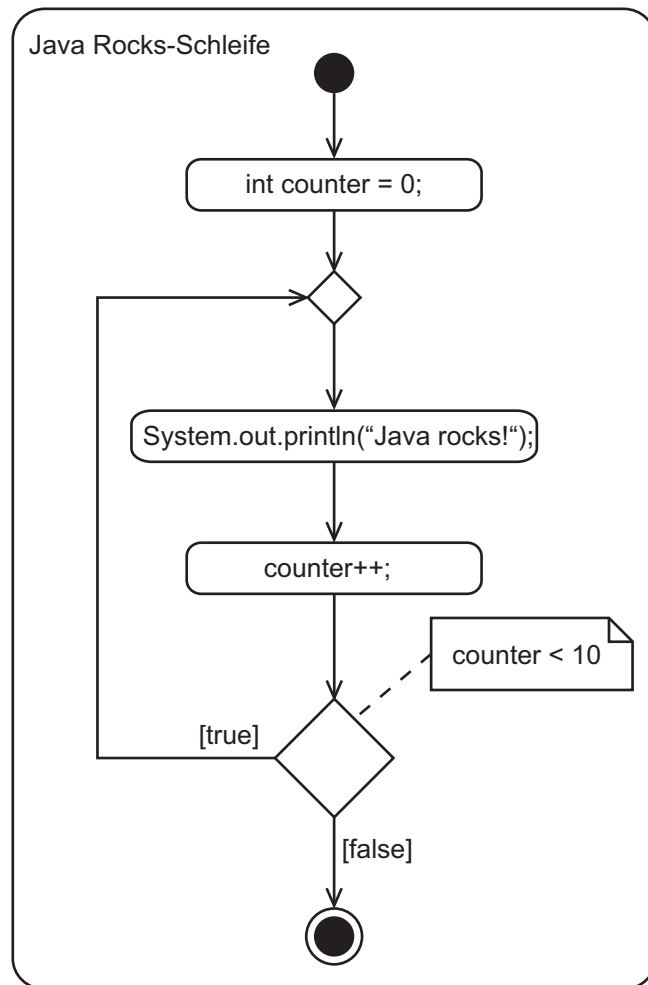
    // beginne eine neue Zeile
    System.out.println();
}
```

### ***Lösung zu Selbsttestaufgabe 14.1-4:***

```
int counter = 0; // bzw. int counter = 10;
do {
    System.out.println("Java rocks!");
    counter++;
} while (counter < 10);
```

Die Schleife wird auch dann einmal durchlaufen, wenn `counter` mit 10 initialisiert wird.

Abb. 14.1-4 zeigt die aus Beispiel 14.1-1 abgeleitete `do`-Schleife.



**Abb. 14.1-4:** Aktivitätsdiagramm zu Beispiel 14.1-1 mit einer `do`-Schleife

Wenn `counter` mit 10 initialisiert wird, wird die Schleifenanweisung dennoch einmal ausgeführt. Erst danach evaluiert der Bedingungsausdruck zu `false`. Beachten Sie, dass das Verhalten der `do`-Schleife mit dem der `while`-Schleife nicht übereinstimmt.

**Lösung zu Selbsttestaufgabe 14.2-1:**

Die `for`-Schleife ist eine abweisende Schleife, da der Rumpf niemals ausgeführt wird, wenn die `booleanExpr` direkt zu Beginn zu `false` ausgewertet wird.

**Lösung zu Selbsttestaufgabe 14.2-2:**

Das folgende Programmstück:

```

forInit;
while (booleanExpr) {
    statement;
    forUpdate;
}
  
```

ist gleichwertig zur schematischen for-Schleife aus Definition 14.2-1. Dies gilt jedoch nur, wenn in statement keine continue-Anweisung (s. Abschnitt 14.3) enthalten ist.

**Lösung zu Selbsttestaufgabe 14.2-3:**

```
int testzahl = 100;
for (int teiler = 1; teiler <= testzahl; teiler++) {
    if (testzahl % teiler == 0) {
        System.out.println(teiler);
    }
}
```

**Lösung zu Selbsttestaufgabe 14.2-4:**

- a) 0
- b) 21
- c) 6
- d) 7
- e) 6

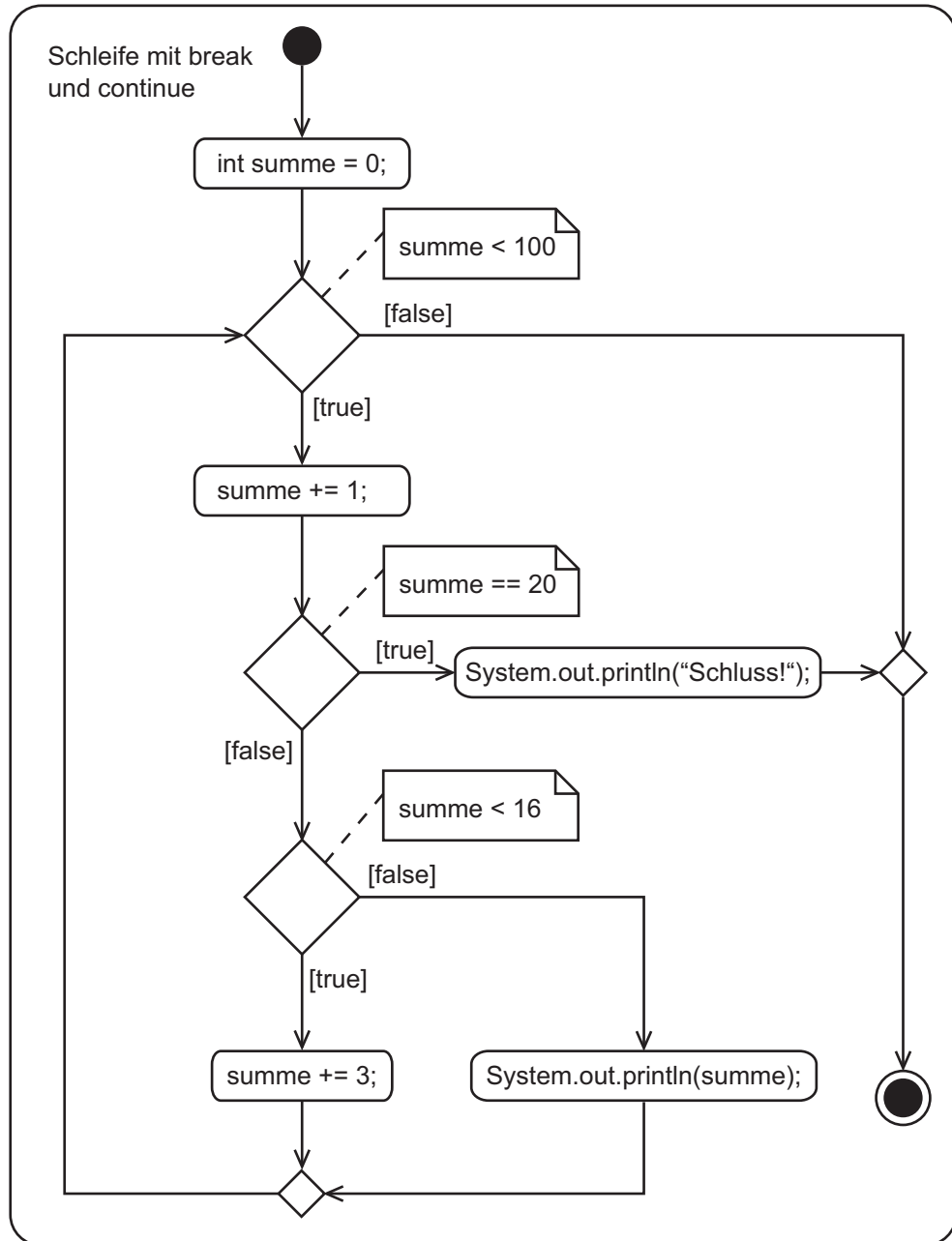
**Lösung zu Selbsttestaufgabe 14.3-1:**

```
byte monat = ...;
switch (monat) {
    case 2:
        System.out.println("28/29 Tage");
        break;
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        System.out.println("31 Tage");
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        System.out.println("30 Tage");
        break;
    default:
        System.out.println("Es gibt keinen solchen Monat");
}
```

**Lösung zu Selbsttestaufgabe 14.3-2:**

Das Programmstück produziert die Ausgaben

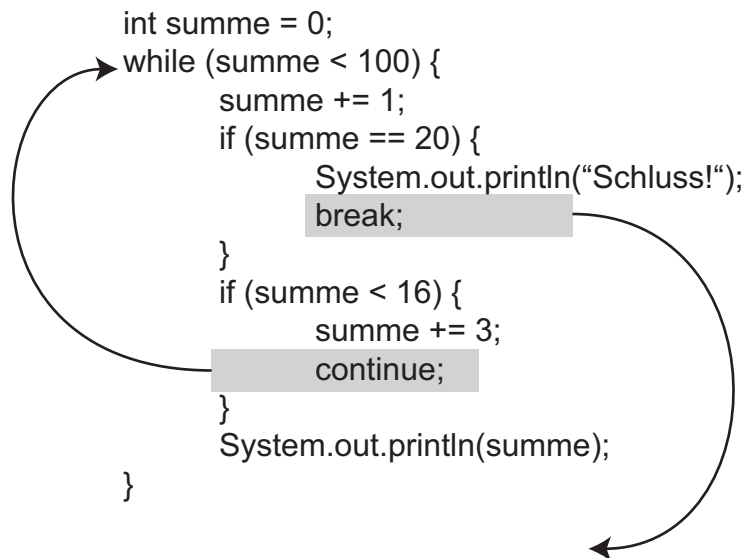
```
17
18
19
Schluss!
```



**Abb. 14.3-3:** Aktivitätsdiagramm zu Beispiel 14.3-4

Das Verhalten des Programmstücks wird in Abb. 14.3-4 graphisch veranschaulicht.





**Abb. 14.3-4:** Schleife mit `break`- und `continue`-Anweisung

**Lösung zu Selbsttestaufgabe 15-1:**

Die Deklaration von `x` in Zeile 5 ist nicht zulässig, da schon in Zeile 1 eine Variable mit dem Namen `x` deklariert wurde. Ebenso ist die Deklaration von `k` in der `for`-Schleife nicht zulässig, da schon in Zeile 3 eine Variable mit diesem Namen deklariert wurde.



## Index

### Symbole

Überlauf 101

Übersetzer-Warnungen 52

Übersetzungsfehler 52

Überwachungsbedingung 41

### A

abweisende Schleife 122

Akteur 11

Aktion 40

Aktivität 40

Aktivitätsdiagramm 40

Aktivitätsendknoten 41

Algorithmus 36

von Euklid 37

Anforderungsanalyse 9

Anpassbarkeit 39

Anweisung 108

einfache 108

Anweisungssequenz 108

Anwendungsfall 10

Beschreibungsschema 17

Anwendungsfallbeschreibung 16

Anwendungsfalldiagramm 12

API 49

arithmetischer Ausdruck 82

Auswertungsreihenfolge 84

arithmetischer Operator 81

Assoziation

im UML-Klassendiagramm 32

zwischen Akteur und Anwendungsfall 12

zwischen Klassen 25

Attribut 24

im UML-Klassendiagramm 31

Attributwert 24

Ausdruck

arithmetischer 82

Auswertungsreihenfolge 84

### B

Block 108

boolean 86

boolean-Literal 86

boolescher Operator 86

break-Anweisung 126

byte 76

Bytecode 50

### C

cast-Operator 104

char 87

char-Literal 88

class-Datei 50

continue-Anweisung 128

### D

Datenlexikon 21

Datentyp 76

boolean 86

byte 76

char 87

double 79

- einfacher **76**
- elementarer **76**
- float **79**
- int **76**
- long **76**
- primitiver **76**
- short **76**
- Dekrementoperator **96**
- Determiniertheit **36**
- Determinismus **36**
- Dezimalliteral **77**
- do-Schleife **121**
- double **79**
- E**
- Effektivität **36**
- einfache Fallunterscheidung **111**
- Enthält-Beziehung **15**
- Entscheidungsknoten **41**
- Erweitert-Beziehung **15**
- erzwungene Typumwandlung **103**
- EVA-Prinzip **46**
- explizite\_Typanpassung **103**
- F**
- Fallunterscheidung
  - einfache **111**
  - Mehrfach- **116**
- final **94**
- finale Variable **94**
- Finitheit **36**
- float **79**
- for-Schleife **122**
- G**
- Gültigkeitsbereich **131**
- Ganze Zahlen **76**
  - Wertebereiche **76**
- Geheimnisprinzip **45**
- Generalisierungsbeziehung
  - in der UML **33**
  - zwischen Akteuren **13**
  - zwischen Anwendungsfällen **15**
  - zwischen Klassen **27**
- Gleitkommazahlen **79**
- H**
- Hexadezimalliteral **77**
- I**
- if-Anweisung **111**
- implizite Typanpassung **98**
- Initialisierung **93**
- Initialknoten **41**
- Inkrementoperator **96**
- int **76**
- int-Literal **77**
- J**
- java-Dateien **50**
- K**
- Klasse **24**
- Klassendiagramm **31**
- Kompatibilitätsregeln **106**
- Komplexität **39**
- Kontrollknoten **40**
- Kontrollstrukturen **40, 110**

Korrektheit **38**

## **L**

Laufzeit-Fehler **52**

Literal **77**

    für Gleitkommazahlen **79**

long **76**

long-Literal **78**

## **M**

main-Methode **52**

Maschinensprache **47**

Mehrfach-Fallunterscheidung **116**

Mehrfachvererbung **28**

Methode **25**

    im UML-Klassendiagramm **31**

Modell **3**

Modellierungssprache **3**

Multiplizitäten **32**

## **N**

Nachricht **28**

Nebeneffekt **96**

nicht abweisende Schleife **122**

## **O**

Oberklasse **27**

Objekt **23**

Objektdiagramm **33**

Oktalliteral **77**

Operator **81**

    arithmetischer **81**

    boolescher **86**

    cast- **104**

Dekrement- **96**

Inkrement- **96**

Rangfolge **83**

Vergleichs- **85**

## **P**

Pflichtenheft **22**

Präzedenzregeln **83**

Programmiersprachen **44**

## **R**

Rückwärtskante **42**

Robustheit **39**

Rolle **11**

## **S**

Schlüsselwörter **90**

Schleife **42**

    abweisende **122**

    do- **121**

    for- **122**

    nicht abweisende **122**

    while- **119**

Schleifenanweisung **119**

Schleifenrumpf **120**

Seiteneffekt **96**

Sequenz **41, 108**

short **76**

Sichtbarkeitsbereich **131**

Softwareentwicklung **2**

Standardfall **116**

Stereotyp **15**

Steuerzeichen **88**

Strukturdiagramm **6**

- strukturierte Sprunganweisung **126**
- Substantivanalyse **30**
- Szenario **17**
  
- T**
- Terminierung **36**
- Typanpassung **98**
  - explizite **103**
  - implizite **98**
- Typeinengung **104**
- Typerweiterung **98**
- Typumwandlung **98**
  - erzwungene **103**
- Typverträglichkeit **98**
  
- U**
- UML **4**
- UML-Aktivitätsdiagramm **40**
- UML-Klassendiagramm **31**
- UML-Objektdiagramm **33**
- Unicode **87**
- Unified Modeling Language **4**
- Unterklasse **27**
  
- V**
- Variable **89**
  - finale **94**
- Variablendeklaration **90**
- Variablenname **90**
- Variablenvereinbarung **90**
- Vereinigungsknoten **41**
- Vererbung **27**
- Vergleichsoperator **85**
- Verhaltensdiagramm **6**
- Verknüpfung **25**
- Verzweigung **41**
- Virtuelle Maschine **49**
- VM **49**
- von Neumann-Architektur **46**
  
- W**
- while-Schleife **119**
- Wiederholung **42**
- Wiederholungsanweisung **119**
- Wiederverwendbarkeit **39**
  
- Z**
- Zeichen **88**
- Zustand **25**
- Zuweisung **92**
  - Auswertung **100**
- Zuweisungsoperator **92**