Kapitel 3

Grundlagen der Programmierung in Java

3.1 Grundelemente eines Java-Programms

Das Erlernen einer Programmiersprache unterscheidet sich im Grunde nicht sonderlich vom Fremdsprachenunterricht in der Schule. Wir haben eine gewisse Grammatik (festgelegt durch den Wortschatz, die Syntax und die Semantik[1]), nach deren Regeln wir Sätze bilden – und eine Unmenge an Vokabeln, die wir für diese Sätze brauchen. Wir formen unsere Sätze aus den gelernten Worten und können diese zu einem komplexeren Gebilde zusammenfügen – beispielsweise einer Geschichte oder einer Bedienungsanleitung für einen Toaster.
In Java (oder einer anderen Programmiersprache) funktioniert das Ganze auf die gleiche Art und Weise. Wir werden lernen, nach gewissen Regeln mit dem Computer zu „sprechen", d. h. ihm verständlich zu machen, was er für uns zu tun hat. Damit uns dies gelingt, müssen wir uns zuerst mit gewissen Grundelementen der Sprache vertraut machen.
Zu diesem Zweck kann man es leider auch nicht ganz vermeiden, sich mit gewissen formalen Aspekten der Sprache zu beschäftigen. Wir werden nach Möglichkeit versuchen, dies mit Hilfe von Beispielen zu tun. In einigen Fällen kommen wir dennoch um eine allgemeine Beschreibungsform nicht herum. Wir werden daher Syntaxregeln (Regeln für zulässige „Sätze" der Sprache Java) in Form eines „Lückentextes" geben. Betrachten wir zunächst ein Beispiel aus dem täglichen Leben:

Syntaxregel

```
In der Cafeteria gab es am «DATUM» «ESSEN» zum «ESSENSART».
```

[1] Zum Verständnis der nachfolgenden Abschnitte ist es ist nicht notwendig, diese Begriffe bereits zu kennen. Angaben dazu finden Sie im Glossar (Anhang C.2).

Die oben angegebene Zeile repräsentiert den formalen Aufbau eines Satzes „Essensbeschreibung“ in der deutschen Sprache. Die Worte «DATUM», «ESSEN» und «ESSENSART» stellen hierbei Lücken bzw. Platzhalter dar, die nach gewissen Regeln gefüllt werden müssen:

- «DATUM» muss ein beliebiges Datum (10.12.2017, 1.8.16), ein Wochentag (Dienstag, Freitag) oder die Worte „heutigen Tag“, „gestrigen Tag“ sein.[2]
- «ESSEN» muss der Name eines beliebigen Essens (Labskaus, Flammkuchen, Rahmblättle, Veggie-Burger, kandierte Schweinsohren mit Ingwersauce) sein.[3]
- «ESSENSART» muss eines der Worte „Frühstück“, „Mittagessen“ oder „Abendbrot“ sein.[4]

Mit obigem Regelwerk können wir nun beliebig gültige Sätze bilden, indem wir die Platzhalter durch ihre gültigen Werte ersetzen. Alles, was außerhalb der Platzhalter steht, wird von uns Punkt für Punkt übernommen:

- In der Cafeteria gab es am Dienstag gebackene Auberginen zum Mittagessen.
- In der Cafeteria gab es am 28.02.18 Toast Helene zum Abendbrot.
- In der Cafeteria gab es am heutigen Tag Ham & Eggs zum Frühstück.

Jeder dieser Sätze stellt eine gültige Essensbeschreibung nach unserer obigen Regel dar. Im Gegenzug hierzu sind die folgenden Sätze falsch, da wir uns nicht an alle Regeln gehalten haben:

- In der Cafeteria gab es am gestern gebackene Auberginen zum Mittagessen. *(„gestern“ ist kein gültiges* «DATUM»*.)*
- In der Cafeteria gab es am 28.02.18 Toast Helene zum Lunch. *(„Lunch“ ist keine gültige* «ESSENSART»*.)*
- Inner Cafeteria gab es am heutigen Tag Ham & Eggs zum Frühstück. *(Es muss „In der“ statt „Inner“ heißen.)*
- In der Cafeteria gab es am Dienstag gebackene Auberginen zum Mittagessen *(Der Punkt am Ende des Satzes fehlt.)*

In unserem Fall ist der Text im Lückentext natürlich keine Umgangssprache, sondern Programmtext, sodass wir Syntaxregeln also in Form von programmähnlichen Texten mit gewissen Platzhaltern angeben werden. Diese Platzhalter werden übrigens fachsprachlich als **Syntaxvariable** bezeichnet. Sind alle aufgestellten Syntaxregeln eingehalten, so sprechen wir auch von einem syntaktisch korrekten Programm. Ist eine Regel verletzt, so ist das Programm syntaktisch nicht korrekt. Ein Programm muss allerdings nicht nur syntaktisch, sondern auch *semantisch* korrekt sein, d. h. seine syntaktischen Elemente müssen auch mit der richtigen Bedeutung verwendet werden. Beispielsweise würde für unseren Platzhalter

[2] Diese Regel ist nicht wirklich eindeutig und lässt viele Möglichkeiten zu einer Präzisierung. Für eine reale Programmiersprache müssen die Angaben genauer sein, damit klar wird, was erlaubt und was verboten ist.

[3] Auch für diese Regel gilt die gerade gemachte Anmerkung.

[4] Durch die eindeutige Festlegung der drei Essensarten ist diese Regel eindeutig.

«DATUM» nur ein Datum in der Vergangenheit als semantisch korrekter Wert in Frage kommen.
Nach diesen vielen Vorerklärungen wollen wir damit beginnen, verschiedene Grundelemente kennenzulernen, die es uns ermöglichen, mit dem Computer in der Sprache Java zu kommunizieren. Diese Abschnitte werden vielen wahrscheinlich etwas langwierig erscheinen; sie stellen jedoch das solide Fundament dar, auf dem unsere späteren Programmierkenntnisse aufbauen!

3.1.1 Kommentare

Wer kennt die Situation nicht? Man hat eine längere Rechnung durchgeführt, einen Löungsvorschlag für ein bestimmtes Problem in einem Zeitschriftenartikel, einem Aufsatz oder einer Skizze erarbeitet – und muss diesen Vorschlag nun anderen Personen erklären. Leider ist die Rechnung, der Artikel oder die Skizze schon ein paar Tage alt, und man erinnert sich nicht mehr an jedes Detail, jeden logischen Schritt. Wie soll man seine Arbeit auf die Schnelle nachvollziehen?
In wichtigen Fällen hat man deshalb bereits beim Erstellen dafür gesorgt, dass jemand anders (oder man selbst) diese Arbeit auch später noch verstehen kann. Hierzu werden Randnotizen, Fußnoten und erläuternde Diagramme verwendet – zusätzliche Kommentare also, die jedoch nicht Bestandteil des eigentlichen Papiers sind.
Auch unsere Programme werden mit der Zeit immer größer. Wir brauchen deshalb eine Möglichkeit, unseren Text mit erläuternden Kommentaren zu versehen. Da sich Textmarker auf dem Monitor jedoch schlecht machen, hat die Sprache Java ihre eigene Art und Weise, mit Kommentaren umzugehen:
Angenommen, wir haben eine Programmzeile verfasst und wollen uns später daran erinnern, was es mit ihr auf sich hat. Die einfachste Möglichkeit wäre, eine Bemerkung oder einen Kommentar direkt in den Programmtext einzufügen. In unserem Beispiel geschieht dies wie folgt:

```
a = b + c;  // hier steht ein Kommentar
```

Sobald der Java-Compiler die Zeichen // in einer Zeile findet, erkennt er einen Kommentar. Alles, was nach diesen beiden Zeichen bis zum Ende der Zeile folgt, geht den Java-Compiler „nichts mehr an“, und es wird beim Übersetzen ignoriert. Der Kommentar kann somit aus allen möglichen Zeichen bestehen, die Java als Eingabezeichen zur Verfügung stellt. Der Kommentar endet also mit dem Ende der Zeile, in der er begonnen wurde.
Manchmal kann es jedoch vorkommen, dass sich Kommentare über mehr als eine Zeile erstrecken sollen. Wir können natürlich jede Zeile mit einem Kommentarzeichen versehen, etwa wie folgt:

```
// Zeile 1
// Zeile 2
// ...
// Zeile n
```

Dies bedeutet jedoch, dass wir auch beim Einfügen weiterer Kommentarzeilen nicht vergessen dürfen, am Anfang jeder Zeile die Kommentarzeichen zu setzen. Java stellt aus diesem Grund eine zweite Form des Kommentars zur Verfügung. Wir beginnen einen mehrzeiligen Kommentar mit den Zeichen `/*` und beenden ihn mit `*/`. Zwischen diesen Zeichen kann ein beliebig langer Text stehen, der natürlich die Zeichenfolge `*/` nicht enthalten darf, da sonst der Kommentar bereits an dieser Stelle beendet wäre:

```
/* Kommentar...
   Kommentar...
   immer noch Kommentar...
   letzte Kommentarzeile...
*/
```

Wir wollen uns bezüglich der Kommentierung unserer Programme zur Angewohnheit machen, möglichst sinnvoll und häufig zu kommentieren. So verlieren wir auch bei einer großen Zahl von Programmen, die wir in einer mindestens ebenso großen Zahl von Dateien speichern müssen, nicht so leicht den Überblick. Um uns auch gleich den richtigen Stil beim Kommentieren von Java-Quelltexten anzugewöhnen, wollen wir uns an das sogenannte **JavaDoc-Format** halten. JavaDoc ist ein sehr hilfreiches Zusatzprogramm, das jedem JDK (Java Development Kit) kostenlos beifügt ist. Mit Hilfe von JavaDoc lassen sich nach erfolgreicher Programmierung automatisch vollständige Dokumentationen zu den erstellten Programmen generieren, was einem im Nachhinein sehr viel Arbeit ersparen kann, sofern man sein Programm fleißig kommentiert hat.
Der Funktionsumfang von JavaDoc ist natürlich wesentlich größer, als wir ihn im Rahmen dieses Abschnitts behandeln können. Es wäre jedoch wenig sinnvoll, auf diesen Punkt näher einzugehen – wir wollen schließlich programmieren lernen! Abgesehen davon wird das Programm ständig weiterentwickelt – interessierten Leserinnen und Lesern sei deshalb die Dokumentation von JavaDoc, die in der Online-Dokumentation eines jeden JDK enthalten ist, wärmstens ans Herz gelegt.
JavaDoc-Kommentare beginnen – ähnlich wie allgemeine Kommentare – stets mit der Zeichenkette `/**` und enden mit `*/`. Es hat sich eingebürgert, zu Beginn jeder Zeile des Kommentars einen zusätzlichen `*` zu setzen, um Kommentare auch optisch vom Rest des Quellcodes abzusetzen.
Ein typischer JavaDoc-Kommentar zu Beginn wäre zum Beispiel folgender:

```
/**
 * Dieses Programm berechnet die Lottozahlen von naechster
 * Woche. Dabei erreicht es im Schnitt eine Genauigkeit
 * von 99,5%.
 *
 * @author Hans Mustermann
 * @version 1.0
 */
```

Werfen wir einmal einen Blick auf die einzelnen Angaben in diesem Beispiel:

- Die ersten Zeilen enthalten eine allgemeine Beschreibung des vorliegenden Programms. Dabei ist vor allem darauf zu achten, dass die Angaben auch ohne

den vorliegenden Quelltext Sinn ergeben sollten, da JavaDoc aus diesen Kommentaren später eigenständige Dateien erzeugt – im Idealfall muss jemand, der die von JavaDoc erzeugten Hilfsdokumente liest, ohne zusätzlichen Blick auf den Quellcode verstehen können, was das Programm macht und wie man es aufruft.

- Als Nächstes sehen wir verschiedene Kommentarbefehle, die stets mit dem Zeichen `@` eingeleitet werden. Hier kann man bestimmte vordefinierte Informationen zum vorliegenden Programm angeben. Dabei spielt die Reihenfolge der Angaben keine Rolle. Auch erkennt JavaDoc natürlich wesentlich mehr Kommentarbefehle als die zwei hier als Beispiel aufgeführten, doch wollen wir uns zunächst auf diese beiden konzentrieren:
  - `@author` der Autor des vorliegenden Programms,
  - `@version` die Versionsnummer des vorliegenden Programms.

3.1.2 Bezeichner und Namen

Wir werden später oft in die Verlegenheit kommen, irgendwelchen Dingen einen **Namen** geben zu müssen – beispielsweise einer Variablen als Platzhalter, um eine Rechnung mit verschiedenen Werten durchführen zu können. Hierzu müssen wir jedoch wissen, wie man in Java solche Namen vergibt.
In ihrer einfachsten Form bestehen Namen aus einem einzigen Bezeichner, der sich in Java aus folgenden Elementen zusammensetzt:

- den **Buchstaben** `a,b,c,...,x,y,z,A,B,C,...,X,Y,Z` des Alphabets (Java unterscheidet also zwischen Groß- und Kleinschreibung).

  Da es sich bei Java um eine internationale Programmiersprache handelt, lässt der Sprachstandard hierbei diverse landesspezifische Erweiterungen zu. So sind etwa japanische Katakana-Zeichen, kyrillische Schrift oder auch die deutschen Umlaute gültige Buchstaben, die in einem Bezeichner verwendet werden dürfen. Wir werden in diesem Skript auf solche Zeichen jedoch bewusst verzichten, da der Austausch derartig verfasster Programme oftmals zu Problemen führt. So werden auf verschiedenen Betriebssystemen die deutschen Umlaute etwa unterschiedlich codiert, sodass Quellcodes ohne einen gewissen Zusatzaufwand nicht mehr portabel sind. Für den übersetzten Bytecode – also die `class`-Dateien, die durch den Compiler `javac` erzeugt werden – ergeben sich diese Probleme nicht. Wir wollen aber auch in der Lage sein, unsere Programmtexte untereinander auszutauschen.
- dem **Unterstrich** „`_`“
- dem **Dollarzeichen** „`$`“
- den **Ziffern** `0,1,2,...,9`

Bezeichner beginnen hierbei immer mit einem Buchstaben, dem Unterstrich oder dem Dollarzeichen – niemals jedoch mit einer Ziffer. Des Weiteren darf kein reserviertes Wort als Bezeichner verwendet werden, d. h. Bezeichner dürfen nicht so lauten wie eine der „Vokabeln“, die wir in den folgenden Abschnitten kennenlernen werden.
Darüber hinaus können sich Namen aber auch aus mehreren Bezeichnern, verbunden durch einen Punkt, zusammensetzen (wie zum Beispiel der Name `System.out.println`).
Folgende Beispiele zeigen gültige Bezeichner in Java:

- `Hallo_Welt`
- `_H_A_L_L_O_`
- `hallo123`
- `hallo_123`

Folgende Beispiele würden in Java jedoch zu einer Fehlermeldung führen:

- `101Dalmatiner` Bezeichner dürfen nicht mit Ziffern beginnen.
- `Das_war's` das Zeichen `'` ist in Bezeichnern nicht erlaubt.
- `Hallo Welt` Bezeichner dürfen keine Leerzeichen enthalten.
- **`class`** dies ist ein reserviertes Wort (siehe Abschnitt 3.1.4).

3.1.3 Literale

Ein **Literal** bzw. eine **Literalkonstante** beschreibt einen konstanten Wert, der sich innerhalb eines Programms nicht ändern kann (und daher vom Java-Compiler normalerweise direkt in den Bytecode aufgenommen wird). Literale haben, abhängig von ihrem Typ (z. B. ganze Zahl oder Gleitkommazahl), vorgeschriebene Schreibweisen. In Java treten folgende Arten von Literalen auf:

- ganze Zahlen (z. B. `23` oder `-166`),[5]
- Gleitkommazahlen (z. B. `3.14`),
- Wahrheitswerte (**`true`** und **`false`**),
- einzelne Zeichen in einfachen Hochkommata (z. B. `'a'`),
- Zeichenketten in Anführungszeichen (z. B. `"Hallo Welt"`) und
- das sogenannte **Null-Literal** für Referenzen, dargestellt durch die Literalkonstante **`null`**.

Wir werden auf die einzelnen Punkte später genauer eingehen. Momentan wollen wir uns nur merken, dass es die sogenannten Literale gibt und sie Teil eines Java-Programms sein können (und werden).

[5] Hierbei zählt das Vorzeichen genau genommen nicht zum Literal; die Negation ist vielmehr eine nachträglich durchgeführte mathematische Operation.

3.1.4 Reservierte Wörter, Schlüsselwörter

Wie bereits erwähnt, gibt es gewisse Wörter, die wir in Java nicht als Bezeichner verwenden dürfen. Zum einen sind dies die Literalkonstanten `true`, `false` und `null`, zum anderen eine Reihe von Wörtern (sogenannte **Wortsymbole**), die in Java mit einer vordefinierten symbolischen Bedeutung belegt sind. Diese werden auch **Schlüsselwörter** genannt. Letztere werden wir nun bald nach und nach in ihrer Bedeutung kennenlernen. Tabelle 3.1 listet diese auf.

Tabelle 3.1: Schlüsselwörter

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

Darüber hinaus gibt es eine Reihe von eingeschränkten Schlüsselwörtern bzw. speziellen Zeichenfolgen, die nur in einem bestimmten Kontext wirklich als Schlüsselwörter dienen. Deren „Geheimnisse“ werden wir erst sehr viel später lüften können. Tabelle 3.2 listet aber auch diese bereits an dieser Stelle unseres Buchs auf.

Tabelle 3.2: Eingeschränkte Schlüsselwörter

`exports`	`module`	`open`	`opens`	`provides`
`requires`	`to`	`transitive`	`uses`	`var`
`with`	`_`			

3.1.5 Trennzeichen

Zu welcher Pferderasse gehören *Blumentopferde*? Heißt es der, die oder das *Kuhliefumdenteich*? Die alten Scherzfragen aus der Vorschulzeit basieren meist auf einem Grundprinzip der Sprachen: Hat man mehrere Wörter, so muss man diese durch Pausen entsprechend voneinander trennen – sonst versteht keiner ihren Sinn! Schreibt man die entsprechenden Wörter nieder, werden aus den Pausen Leerzeichen, Gedankenstriche und Kommata.
Auch der Java-Compiler muss in der Lage sein, einzelne Bezeichner und Wortsymbole voneinander zu trennen. Hierzu stehen uns mehrere Möglichkeiten zur Verfügung, die sogenannten Trennzeichen. Diese sind:

- Leerzeichen
- Zeilenendezeichen (der Druck auf die ENTER-Taste)
- Tabulatorzeichen (die TAB-Taste)
- Kommentare
- Operatoren (wie zum Beispiel + oder *)
- die Interpunktionszeichen . , ;) ({ } [ ]

Die beiden letztgenannten Gruppen von Zeichen haben in der Sprache Java jedoch eine besondere Bedeutung. Man sollte sie deshalb nur dort einsetzen, wo sie auch hingehören.
Unmittelbar aufeinander folgende Wortsymbole, Literale oder Bezeichner müssen durch mindestens eines der obigen Symbole voneinander getrennt werden, sofern deren Anfang und Ende nicht aus dem Kontext erkannt werden kann. Hierbei ist es in Java eigentlich egal, welche Trennzeichen man verwendet (und wie viele von ihnen). Steht im Programm zwischen zwei Bezeichnern beispielsweise eine Klammer, so gilt diese bereits als Trennsymbol. Wir müssen keine weiteren Leerzeichen einfügen (können und sollten dies aber tun). Die Programmzeile

```
public static void main (String[] args)
```

wäre demnach völlig äquivalent zu folgenden Zeilen:

```
public // verwendete Trennsymbole: Leerzeichen und Kommentar
  static
/*Verwendung eines Zeilenendezeichens*/
                 void
              //..
             // ..
// man kann auch mehrere Zeilenvorschuebe verwenden
  main(// hier sind nun zwei Trennzeichen: Klammer und Kommentar!
                 String[] args)
```

Übersichtlicher wird der Text hierdurch jedoch nicht. Wir werden uns deshalb später auf einige Konventionen einigen, um unsere Programme lesbarer zu machen.

3.1.6 Interpunktionszeichen

Dem Punkt in der deutschen Sprache entspricht in Java das Semikolon. Befehle (sozusagen die Sätze der Sprache) werden in Java immer mit einem Semikolon abgeschlossen. Fehlt dieses, liefert der Übersetzer eine Fehlermeldung der Form

Konsole

```
Fehlerhaft.java:10: ';' expected.
```

Hierbei ist `Fehlerhaft.java` der Dateiname, unter dem das Programm gespeichert wurde. Die angegebene Zahl steht für die Nummer der Zeile, in der der Fehler aufgetreten ist.

Wie bereits erwähnt, existieren in Java neben dem Semikolon weitere Interpunktionszeichen. Werden z. B. mehrere Befehle zu einem **Block** zusammengefasst, so geschieht dies, indem man vor den ersten und hinter den letzten Befehl eine geschweifte Klammer setzt. Hierzu ein Beispiel (mehr dazu in Abschnitt 3.5.2):

```
{                              // Blockbeginn
  System.out.println("B1"); //   Befehl Nr. 1
  System.out.println("B2"); //   Befehl Nr. 2
  System.out.println("B3"); //   Befehl Nr. 3
}                              // Blockende
```

3.1.7 Operatorsymbole

Operatoren sind spezielle Symbole, die dazu dienen, jeweils bis zu drei unterschiedliche Ausdrücke – die sogenannten Operanden – zu einem neuen Ausdruck zu verknüpfen. Wir unterscheiden die Operatoren nach der Anzahl ihrer Operanden:

monadische Operatoren sind Operatoren, die nur einen Operanden benötigen. Beispiele hierfür sind die Operatoren `++` oder `--`. Solche Operatoren werden auch als **einstellige** oder **unäre** Operatoren bezeichnet.

dyadische Operatoren verknüpfen zwei Operanden und sind die am häufigsten vorkommende Art von Operatoren. Beispiele hierfür sind etwa die Operatoren `+`, `-` oder `==`. Solche Operatoren werden auch **zweistellige** oder **binäre** Operatoren genannt.

triadische Operatoren verknüpfen drei Operanden. Einziges Beispiel in Java ist der Operator `?:` (Fragezeichen-Doppelpunkt). Er wird auch als **dreistelliger** oder **ternärer** Operator bezeichnet.

Operatoren sind in Java mit sogenannten Prioritäten versehen, wodurch in der Sprache gewisse Reihenfolgen in der Auswertung (zum Beispiel Punkt- vor Strichrechnung) festgelegt sind. Wir werden uns mit diesem Thema an späterer Stelle befassen und dann auch klären, wie bei der Ausführung von Programmen solche Ausdrücke zum jeweiligen Zeitpunkt des Ablaufs in aktuelle Berechnungen umgesetzt werden.

3.1.8 `import`-Anweisungen

Viele Dinge, die wir in Java benötigen, befinden sich nicht im Kern der Sprache – beispielsweise die Bildschirmausgabe oder mathematische Standardfunktionen wie Sinus oder Cosinus. Sie wurden in sogenannte Klassen ausgelagert, die erst beim Programmstart bei Bedarf hinzugeladen werden müssen. Der Übersetzer muss sie namentlich kennen. Einige dieser Klassen kennt er ohne unser Zutun (sie liegen in einem Standardpaket namens `java.lang`). Andere wiederum müssen wir ihm explizit namentlich bekannt machen, indem wir sie **importieren** lassen.

Um den Übersetzer anzuweisen, einen solchen Vorgang einzuleiten, wird eine sogenannte **import**-Anweisung verwendet. Diese macht dem Compiler die von der Anweisung bezeichnete Klasse zugänglich, d. h. er kann auf sie zugreifen und sie verwenden. Beispiele hierfür sind etwa die Klasse `Scanner` aus dem Paket `java.util` oder die Klasse `IOTools`, mit deren Hilfe Sie später Eingaben von der Tastatur bewerkstelligen werden. Letztgenannte Klasse gehört zu einem frei verfügbaren Paket namens `Prog1Tools`, das nicht von der Firma Sun bzw. Oracle stammt und somit nicht zu den standardmäßig eingebundenen Werkzeugen gehört. Wenn Sie dieses Paket auf Ihrem Rechner installiert haben und die Klasse in einem Ihrer Programme verwenden wollen (siehe Abschnitt 3.4.4), beginnen Sie Ihr Programm mit

```
import Prog1Tools.IOTools;
```

oder mit

```
import static Prog1Tools.IOTools.*;
```

wobei die zweite Variante einen statischen Import durchführt. Was genau es damit auf sich hat, werden wir in den Abschnitten 3.4.4.1 und 5.4.3 näher erläutern.
Viele Klassen, die wir vor allem zu Beginn benötigen, werden vom System automatisch als bekannt vorausgesetzt – es wird also noch eine Weile dauern, bis in unseren Programmen die **import**-Anweisung zum ersten Mal auftaucht. Sie ist dennoch das Grundelement eines Java-Programms und soll an dieser Stelle deshalb schon erwähnt werden.

3.1.9 Zusammenfassung

Wir haben in diesem Abschnitt die verschiedenen Komponenten kennengelernt, aus denen sich ein Java-Programm zusammensetzt. Wir haben gelernt, dass es aus

- Kommentaren,
- Bezeichnern,
- Trennzeichen,
- Wortsymbolen,
- Interpunktionszeichen,
- Operatoren und
- **import**-Anweisungen

bestehen kann, auch wenn nicht jede dieser Komponenten in jedem Java-Programm auftaucht. Wir haben gelernt, dass es wichtig ist, seine Programme gründlich zu dokumentieren, und haben uns auf einige einfache Konventionen festgelegt, mit denen wir einen ersten Schritt in diese Richtung tun wollen. Mit diesem Wissen können wir beginnen, erste Java-Programme zu schreiben.

3.1.10 Übungsaufgaben

Aufgabe 3.1

Die Zeichenfolge `dummy` hat in Java keine vordefinierte Bedeutung – sie wird also, wenn sie ohne besondere Vereinbarungen im Programmtext steht, zu einem Compilierungsfehler führen. Welche der folgenden Zeilen könnte einen solchen Fehler verursachen?

```
      dummy
      dummy;
      dummy; //
//    dummy
//    dummy;
/*    dummy */
*/    dummy /*
/**/ dummy
      dummy /* */
```

Aufgabe 3.2

Die nachfolgenden Zeilen sollen jeweils einen Bezeichner enthalten. Welche Zeilen sind unzulässig?

```
Karl der Grosse
Karl_der_Grosse
Karl,der_Grosse
0 Ahnung?
0_Ahnung
null_Ahnung!
1234abc
_1234abc
_1_2_3_4_abc
```

Aufgabe 3.3

Geben Sie das folgende Programm in Ihren Computer ein:

```
1 /* Beispiel: berechnet 7 + 11 */
2 public clss Berechnung {
3   public static void main (String[] args) {
4     int sume;
5     summe = 7 + 13;
6     System.out.print("7 + 11 ergibt");
7     System.out.println(summe)
8   }
9 }
```

Finden Sie die kleinen Fehler, die sich in das Programm geschlichen haben, indem Sie das Programm compilieren und aus den Fehlermeldungen auf den jeweiligen Fehler im Programm schließen.

Das Programm hat auch einen kleinen Fehler, den der Compiler nicht finden wird. Wenn Sie das Programm starten, können Sie aber (mit etwas Grundschulwissen) auch diesen Fehler entdecken und korrigieren!

3.2 Erste Schritte in Java

Nachdem wir nun über die Grundelemente eines Java-Programms Bescheid wissen, können wir damit beginnen, unsere ersten kleineren Programme in Java zu schreiben. Wir werden mit einigen einfachen Problemstellungen beginnen und uns langsam an etwas anspruchsvollere Aufgaben herantasten.
Wenn wir ein Programmm (auch **Applikation** genannt) schreiben, geben wir dem Computer in einer Aneinanderreihung von diversen Befehlen an, was er genau zu tun hat. Diese Befehle werden (nach dem Übersetzen durch den Java-Compiler) mit dem Java-Interpreter gestartet und abgearbeitet. Solche Applikationen können auch grafische und interaktive Elemente beinhalten, müssen es aber nicht.

3.2.1 Grundstruktur eines Java-Programms

Angenommen, wir wollen ein Programm schreiben, das wir `Hallo Welt` nennen. Als Erstes müssen wir uns darüber klar werden, dass dieser Name kein Bezeichner ist – Leerzeichen sind nicht erlaubt. Wir entfernen deshalb das Leerzeichen und erstellen mit unserem Editor eine Datei mit dem Namen `HalloWelt.java`.
Hinweis: Es gibt Fälle, in denen die Datei nicht wie das Programm heißen muss. Im Allgemeinen erwartet dies der Übersetzer jedoch; wir wollen es uns deshalb von Anfang an angewöhnen.
Geben wir nun in unseren Editor folgende Zeilen ein (die Kommentare können auch wegfallen):

```
// Klassen- bzw. Programmbeginn
public class HalloWelt {
  // Beginn des Hauptprogramms
  public static void main(String[] args) {
    // HIER STEHT EINMAL DAS PROGRAMM...
  } // Ende des Hauptprogramms
} // Ende des Programms
```

Wir sehen, dass das Programm aus zwei Ebenen besteht, die wir an dieser Stelle nochmals kurz erklären wollen:

- Mit der Zeile

  ```
  public class HalloWelt {
  ```

  machen wir dem Übersetzer klar, dass die folgende Klasse[6] den Namen `HalloWelt` trägt. Diese Zeile darf niemals fehlen, denn wir müssen unse-

[6] Ein Begriff aus dem objektorientierten Programmieren; wir wollen ihn im Moment mit Programm gleichsetzen.

rem Programm natürlich einen Namen zuweisen. Der Übersetzer speichert das compilierte Programm nun in einer Datei namens `HalloWelt.class` ab.

- Es ist prinzipiell möglich, ein Programm in mehrere Abschnitte zu unterteilen. Der Programmablauf wird durch die sogenannte **Hauptmethode** (oder auch `main`-Methode), also quasi das Hauptprogramm, gesteuert. Diese wird mit der Zeile

```
public static void main(String[] args) {
```

eingeleitet. Der Übersetzer erfährt somit, an welcher Stelle er später die Ausführung des Programms beginnen soll.

Es fällt auf, dass beide Zeilen jeweils mit einer geschweiften Klammer enden. Wir erinnern uns: Dieses Interpunktionszeichen steht für den Beginn eines Blocks, d. h. es werden mehrere Zeilen zu einer Einheit zusammengefasst. Diese Zeichen entbehren nicht einer gewissen Logik. Der erste Block fasst die folgenden Definitionen zu einem Block zusammen; er weist den Compiler an, sie als eine Einheit (eben das Programm bzw. die Klasse) zu betrachten. Der zweite Block umgibt die Anweisungen der Hauptmethode.
Achtung: Jede geöffnete Klammer (Blockanfang) muss sich irgendwann auch wieder schließen (Blockende). Wenn wir dies vergessen, wird das vom Compiler mit einer Fehlermeldung bestraft!

3.2.2 Ausgaben auf der Konsole

Wir wollen nun unser Programm so erweitern, dass es die Worte `Hallo Welt` auf dem Bildschirm, also auf unserem Konsolenfenster, ausgibt. Wir ersetzen hierzu mit unserem Editor die Zeile

```
// HIER STEHT EINMAL DAS PROGRAMM...
```

durch die Zeile

```
System.out.println("Hallo Welt");
```

Wir sehen, dass sich diese Zeile aus mehreren Komponenten zusammensetzt.

- Die Anweisung `System.out.println(...)` weist den Computer an, etwas auf dem Bildschirm auszugeben. Das Auszugebende muss zwischen den runden Klammern stehen.
- Der Text `"Hallo Welt"` stellt das Ausgabeargument dar, entspricht also dem auf dem Bildschirm auszugebenden Text. Die Anführungszeichen tauchen bei der Ausgabe nicht auf. Sie markieren nur den Anfang und das Ende des Textes.
- Das Interpunktionszeichen „`;`“ muss jede Anweisung (jeden Befehl) beenden. Ein vergessenes Semikolon ist wohl der häufigste Programmierfehler und unterläuft selbst alten Hasen hin und wieder.

Wir speichern unser so verändertes Programm ab und geben in der Kommandozeile die Anweisung

Konsole
```
javac HalloWelt.java
```

ein. Der Compiler übersetzt unser Programm nun in den interpretierbaren Bytecode.[7] Das Ergebnis der Übersetzung finden wir unter dem Namen `HalloWelt.class` wieder.
Nun wollen wir unser Programm mit Hilfe des Interpreters starten. Hierzu geben wir in der Kommandozeile ein:[8]

Konsole
```
java HalloWelt
```

Unser Programm wird tatsächlich ausgeführt – die Worte `Hallo Welt` erscheinen auf dem Bildschirm. Ermutigt von diesem ersten Erfolg, erweitern wir unser Programm um die Zeile

```
System.out.println("Mein erstes Programm :-)");
```

und übersetzen erneut. Starten wir das Programm, lautet die Ausgabe auf dem Bildschirm nun

Konsole
```
Hallo Welt
Mein erstes Programm :-)
```

Wir sehen, dass der erste Befehl `System.out.println(...)` nach dem ausgegebenen Text einen Zeilenvorschub durchführt. Was ist jedoch, wenn wir dies nicht wollen?
Die einfachste Möglichkeit ist, beide Texte in einer Anweisung zu drucken. Die neue Zeile würde dann entweder

```
System.out.println("Hallo Welt Mein erstes Programm :-)");
```

oder

```
System.out.println("Hallo Welt " + "Mein erstes Programm :-)");
```

heißen. Letztere Zeile enthält einen für uns neuen Operator. Das Zeichen + addiert nicht etwa zwei Texte (wie sollte dies auch funktionieren?). Es weist Java vielmehr an, die Texte `Hallo Welt` und `Mein erstes Programm :-)` unmittelbar aneinanderzuhängen. Wir werden diesen Operator später noch zu schätzen wissen.
Eine weitere Möglichkeit, das gleiche Ergebnis zu erzielen, ist die Verwendung des Befehls `System.out.print`. Im Gegensatz zu `System.out.println` wird nach der Ausführung nicht in die nächste Bildschirmzeile gewechselt. Weitere

[7] Wir erinnern uns: Bei Verwendung einer integrierten Java-Entwicklungsumgebung wie zum Beispiel *NetBeans*, *Eclipse* oder *IntelliJ IDEA* müssen wir den Übersetzungsvorgang meist gar nicht explizit auslösen, da er in der Regel automatisch abläuft.

[8] Wir erinnern uns: Falls wir eine integrierte Java-Entwicklungsumgebung wie zum Beispiel *NetBeans*, *Eclipse* oder *IntelliJ IDEA* einsetzen, können wir direkt den Start- oder Run-Knopf (▷) der Entwicklungsumgebung betätigen, wonach sich prinzipiell automatisch ein Konsolenfenster öffnet, in dem unser Programm mit Hilfe des Java-Interpreters ausgeführt wird.

Zeichen werden noch in die gleiche Zeile geschrieben. Unser so verändertes Programm sähe wie folgt aus:

```
1 // Klassen- bzw. Programmbeginn
2 public class HalloWelt {
3   // Beginn des Hauptprogramms
4   public static void main(String[] args) {
5     System.out.print("Hallo Welt ");
6     System.out.println("Mein erstes Programm :-)");
7   }  // Ende des Hauptprogramms
8 }  // Ende des Programms
```

3.2.3 Eingaben von der Konsole

Für den Anfänger wäre es sicher wünschenswert, wenn die Eingabe von der Konsole, also von der Tastatur, genauso einfach realisiert werden könnte wie die im letzten Abschnitt beschriebene Ausgabe. Leider ist dem nicht so, da die Entwickler von Java diese Art von „Einfachst-Eingabe“ in allgemeiner Form implementiert haben. Um diese Konzepte verstehen und anwenden zu können, muss man eigentlich schon Kenntnisse über das objektorientierte Programmieren und die sogenannten I/O-Streams, über die Ein- und Ausgaben realisiert sind, haben. Es steht in der Java-Klassenbibliothek zwar mit der Klasse `Scanner`[9] eine Klasse bereit, die für eine vereinfachte Konsoleneingabe genutzt werden kann, aber auch ihr Einsatz erfordert zumindest ein Grundverständnis der Konzepte, die im Umgang mit Objekten (Strom- bzw. `Scanner`-Objekte) zur Anwendung kommen. Insbesondere Programmieranfänger haben daher große Schwierigkeiten, die Konsoleneingabe zu benutzen und vollständig zu verstehen, weil sie mit speziellen Klassen und Methoden aus dem Eingabestrom von der Tastatur z. B. ganzzahlige Werte oder Gleitkommazahlen extrahieren müssen.
Um dem Abhilfe zu schaffen, wurden für die Anfängerkurse der Autoren die `IOTools` geschrieben und den Kursteilnehmern zur Verfügung gestellt. Auf der Website zu diesem Buch [47] stehen die `IOTools` (im Rahmen des Pakets `Prog1Tools`) zum Download zur Verfügung. Eine detaillierte Beschreibung der Klasse `IOTools` und ihrer Methoden findet sich auch im Anhang C. Prinzipiell kann an dieser Stelle gesagt werden, dass es für jede Art von Wert (ganze Zahl, Gleitkommawert, logischer Wert etc.), der eingelesen werden soll, eine entsprechende Methode gibt. Die Methode `readInteger` liest beispielsweise eine ganze Zahl von der Tastatur ein. Wenn wir die in Java verfügbaren Datentypen kennengelernt haben, werden wir auf deren Eingabe nochmals zurückkommen.

3.2.4 Schöner programmieren in Java

Wir haben bereits gesehen, dass Programme sehr strukturiert und übersichtlich gestaltet werden können – oder unstrukturiert und chaotisch. Wir wollen uns des-

[9] Wir werden uns in Abschnitt 17.3.5.2 näher mit dieser Klasse beschäftigen.

halb an einige „goldene Regeln“ halten, mit denen wir unser Programme übersichtlicher und lesbarer gestalten können.

1. *Niemals mehr als einen Befehl in eine Zeile schreiben!* Auf diese Art und Weise können wir beim späteren Lesen des Codes auch keine Anweisung übersehen.
2. Wenn wir einen neuen Block beginnen, *rücken* wir alle in diesem Block stehenden Zeilen um einige (beispielsweise zwei) Zeichen *nach rechts ein*. Wir haben auf diese Art und Weise stets den Überblick darüber, wie weit ein Block eigentlich reicht.
3. *Das Blockendezeichen „}“ wird stets so eingerückt*, dass es mit der Einrückung der Zeile übereinstimmt, in der der Block geöffnet wurde. Wir können hierdurch vergessene Klammern sehr viel schneller aufspüren.

Wir wollen dies an einem Beispiel verdeutlichen, indem wir auf das nachfolgende noch „unschöne“ Programm unsere obigen Regeln anwenden.

```
1 public class Unsorted {public static void main(String[] args) {
2 System.out.print("Ist dieses");System.out.
3 print(" Programm eigentlich");System.out.println(" noch "
4 +"lesbar?");}}
```

Obwohl es nur aus wenigen Zeilen besteht, ist das Lesen dieses kurzen Programms doch schon recht schwierig. Wie sollen wir später mit solchen Texten zurechtkommen, die sich aus einigen *hundert* Zeilen Programmcode zusammensetzen? Wir spalten das Programm deshalb gemäß unseren Regeln auf und rücken entsprechend ein:

```
1 public class Unsorted {
2   public static void main(String[] args) {
3     System.out.print("Ist dieses");
4     System.out.print(" Programm eigentlich");
5     System.out.println(" noch " + "lesbar?");
6   }
7 }
```

Übersetzen wir das Programm und starten es, so können wir die auf dem Bildschirm erscheinende Frage eindeutig bejahen. Weitere Tipps und Tricks für „schönes Programmieren“ haben wir im Anhang A zusammengefasst.

3.2.5 Zusammenfassung

Wir haben die Grundstruktur einer Java-Applikation kennengelernt und erfahren, wie man mit den Befehlen `System.out.println` und `System.out.print` Texte auf dem Bildschirm ausgibt. Hierbei wurde auch der Operator + erwähnt, der Texte aneinanderfügen kann. Dieses Wissen haben wir angewendet, um unser erstes Java-Programm zu schreiben.
Außerdem haben wir uns auf einige Regeln geeinigt, nach denen wir unsere Programme formatieren wollen. Wir haben gesehen, wie die einfache Anwendung dieser „Gesetze“ unseren Quelltext viel besser lesbar macht.

3.2.6 Übungsaufgaben

Aufgabe 3.4

Schreiben Sie ein Java-Programm, das Ihren Namen dreimal hintereinander auf dem Bildschirm ausgibt.

Aufgabe 3.5

Gegeben ist folgendes Java-Programm:

```
1                  public class Strukturuebeung
2    { public static void main (String[] args){
3  System.out.println("Was mag ich wohl tun?")}
```

Dieses Programm enthält zwei Fehler, die es zu finden gilt. Versuchen Sie es zuerst anhand des vorliegenden Textes. Wenn Ihnen dies nicht gelingt, formatieren Sie das Programm gemäß unseren „goldenen Regeln“. Versuchen Sie auch einmal, das fehlerhafte Programm zu übersetzen. Machen Sie sich mit den auftretenden Fehlermeldungen vertraut.

3.3 Einfache Datentypen

Natürlich reicht es uns nicht aus, einfache Meldungen auf dem Bildschirm auszugeben (dafür bräuchten wir keine Programmiersprache). Wir wollen Java benutzen können, um gewisse Effekte zu erzielen, Berechnungen auszuführen, Abläufe zu automatisieren oder Probleme des elektronischen Handels (z. B. die sichere Übertragung von Daten über das Internet oder die Analyse von großen Datenmengen) zu lösen, also allgemein Algorithmen zu realisieren und Daten zu verarbeiten. Wir wollen uns aus diesem Grund zunächst darüber klar werden, wie man in Java mit einfachen Daten (Zahlen, Buchstaben usw.) umgehen kann. Dazu werden wir uns erst einmal die Wertebereiche der einfachen Datentypen anschauen. Bei der Definition eines Datentyps werden der Wertebereich, d. h. die möglichen Werte dieses Typs, und die für diese Werte zugelassenen Grundoperationen festgelegt. Nachfolgend wollen wir eine erste Einführung in die einfachen Datentypen von Java geben. Diese heißen deshalb einfach, weil es neben ihnen auch kompliziertere Datentypen gibt, die sich auf eine noch festzulegende Weise aus einfachen Datentypen zusammensetzen.

3.3.1 Ganzzahlige Datentypen

Wie geht ein Computer mit ganzen Zahlen um? Er speichert sie als eine Folge von binären Zeichen (also 0 oder 1), die ganzzahlige Potenzen der Zahl 2 repräsentieren. Die Zahl 23 kann etwa durch die Summe

$$1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

ausgedrückt werden, die dann im Speicher eines Rechners durch die Binärfolge `10111` (also mit 5 Stellen) codiert werden kann. Negative Zahlen lassen sich durch verschiedene Methoden codieren, auf die wir hier jedoch nicht im Detail eingehen werden. In jedem Fall benötigt man jedoch eine weitere Stelle für das Vorzeichen. Nehmen wir an, wir haben 1 Byte – dies sind 8 Bits, also 8 binäre Stellen – zur Verfügung, um eine Zahl darzustellen. Das erste Bit benötigen wir für das Vorzeichen. Die größte Zahl, die wir mit den noch verbleibenden 7 Ziffern darstellen können, ist somit

$$1 \cdot 64 + 1 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 127.$$

Aufgrund der rechnerinternen Darstellung negativer Zahlen (der sogenannten Zweierkomplement-Darstellung) ist die kleinste negative Zahl betragsmäßig um 1 größer, d. h. in diesem Fall -128. Wir können mit 8 Bits also $127 + 128 + 1 = 256$ verschiedene Zahlen darstellen. Hätten wir mehr Stellen zur Verfügung, würde auch unser Zahlenvorrat wachsen.
Der Datentyp **`byte`** repräsentiert in Java genau diese Darstellung. Eine Zahl vom Typ **`byte`** ist 8 Bits lang und liegt im Bereich von -128 bis $+127$. Im Allgemeinen ist dieser Zahlenbereich viel zu klein, um damit vernünftig arbeiten zu können. Java besitzt aus diesem Grund weitere Arten von ganzzahligen Datentypen, die zwei, vier oder acht Byte lang sind. Tabelle 3.3 fasst diese Datentypen zusammen.

Tabelle 3.3: Ganzzahlige Datentypen

Typname	größter Wert	kleinster Wert	Länge
`byte`	127	-128	8 Bits
`short`	32767	-32768	16 Bits
`int`	2147483647	-2147483648	32 Bits
`long`	9223372036854775807	-9223372036854775808	64 Bits

Wie wir bereits in Abschnitt 3.1.3 gesehen haben, bestehen Literalkonstanten für die ganzzahligen Datentypen einfach aus einer Folge von Ziffern (0 bis 9). Für solche Konstanten wird standardmäßig angenommen, dass es sich um eine 32-Bit-Zahl (also eine **`int`**-Konstante) handelt. Wollen wir stattdessen mit 64 Bits arbeiten (also mit einer Zahl im **`long`**-Format), müssen wir an die Zahl die Endung `L` anhängen.
Wir wollen dies an einem Beispiel verdeutlichen. Die Befehle

```
System.out.println
```

und

```
System.out.print
```

sind auch in der Lage, einfache Datentypen wie die Ganzzahlen auszudrucken. Wir versuchen nun, die Zahl `9223372036854775807` auf dem Bildschirm auszugeben. Hierzu schreiben wir folgendes Programm:

```
1 public class Longtst {
2   public static void main (String[] args) {
```

```
3       System.out.println(9223372036854775807);
4     }
5   }
```

Rufen wir den Compiler mit dem Kommando `javac Longtst.java` auf, so erhalten wir folgende Fehlermeldung:

Konsole

```
Longtst.java:3: integer number too large: 9223372036854775807
    System.out.println(9223372036854775807);
                       ^
1 error
```

In Zeile 3 der Datei `Longtst.java` haben wir also eine Zahl verwendet, die zu groß ist. Zu groß deshalb, weil sie ja standardmäßig als **`int`**-Wert angenommen wird, aber laut Tabelle 3.3 deutlich größer als `2147483647` ist und somit nicht mehr mit 32 Bits dargestellt werden kann. Eine derartige Zahl muss explizit als längere Zahl gekennzeichnet werden! Wir ändern die Zeile deshalb wie folgt:

```
System.out.println(9223372036854775807L);
```

Durch die Hinzunahme der Endung `L` wird die Zahl als eine **`long`**-Zahl betrachtet und somit mit 64 Bits codiert (in die sie laut Tabelle gerade noch hineinpasst). Der entsprechende Datentyp heißt in Java **`long`**.

3.3.1.1 Literalkonstanten in anderen Zahlensystemen

Neben der rein dezimalen Schreibweise von ganzzahligen Werten können Literalkonstanten in Java auch als binäre Zahlen (Zahlen im Zweier-System) als oktale Zahlen (Zahlen im Achter-System) oder als hexadezimale Zahlen (Zahlen im Sechzehner-System) geschrieben werden. Binäre Zahlen müssen mit einem Präfix der Form `0b` oder `0B` beginnen und dürfen danach nur aus den Ziffern 0 und 1 bestehen. Oktale Zahlen müssen mit einer führenden `0` beginnen und dürfen nur Ziffern im Bereich 0 bis 7 enthalten. Hexadezimale Zahlen müssen mit `0x` beginnen und dürfen Ziffern im Bereich 0 bis 9 und A bis F enthalten. Die vier **`int`**-Konstanten `42`, `052`, `0x2A` und `0b101010` sind somit alternative Schreibweisen für den ganzzahligen dezimalen Wert 42. Dementsprechend wird von unserem Beispielprogramm

```
1  public class BinaereLiterale {
2    public static void main(String[] args) {
3      // Binaere Literale
4      int dezWert = 42;       // Die Zahl 42, dezimal
5      int oktWert = 052;      // Die Zahl 42, oktal
6      int hexWert = 0x2A;     // Die Zahl 42, hexadezimal
7      int binWert = 0b101010; // Die Zahl 42, binaer
8      System.out.println(dezWert + ", " + oktWert + ", " +
9                         hexWert + " und nochmal " + binWert);
10   }
11 }
```

auch die Bildschirmausgabe

Konsole

```
42, 42, 42 und nochmal 42
```

erzeugt.

3.3.1.2 Unterstrich als Trennzeichen in Literalkonstanten

Wenn in einem Java-Programm numerische Konstanten benötigt werden, die aus vielen Ziffern bestehen, ist es manchmal ziemlich schwer, die Größenordnung der jeweiligen Teilziffernfolge zu erkennen. Beispielsweise kann man der üblichen Schreibweise der Konstante 2147483647 auf den ersten Blick schwer entnehmen, ob es sich hier um 214 Millionen oder 2,14 Milliarden handelt. In hexadezimalen Werten ist es nicht weniger schwer, Byte-Portionen zu erkennen. So kann man in der Notation FFECDE5E schwerlich auf Anhieb feststellen, ob nun FE oder EC als Byte zu interpretieren ist.
Aus diesem Grund kann der Unterstrich als Trennzeichen innerhalb von Ziffernfolgen gesetzt werden, ohne dass dadurch der Wert der Konstante verändert wird. Für unsere obigen Beispiele könnten wir daher nun in Java auch `2_147_483_647` bzw. `FF_EC_DE_5E` notieren. Der Einsatz des Unterstrichs ist dabei nicht auf ganzzahlige Werte beschränkt. Auch **`float`**- und **`double`**-Konstanten können damit notiert werden. Unser Beispielprogramm

```
1  public class Unterstrich {
2    public static void main(String[] args) {
3      //  Unterstrich in numerischen Literalen
4      long creditKartenNummer = 1234_5678_9012_3456L;
5      long versicherungsNummer = 999_99_9999L;
6      float pi =   3.14_15F;
7      double e =   2.71_82_81_82_84_59;
8      long hexBytes = 0xFF_EC_DE_5E;
9      long hexText  = 0xCAFE_BABE;
10     long maxLong = 0x7fff_ffff_ffff_ffffL;
11     long binBytes = 0b11010010_01101001_10010100_10010010;
12     int ok1 = 4_2;
13     int ok2 = 4_______2;
14     int ok3 = 0x2__a;
15     int ok4 = 0_42;
16     int ok5 = 04_2;
17   }
18 }
```

demonstriert einige Möglichkeiten für solche mit Unterstrichen lesbarer gestaltete Literalkonstanten.
Allerdings gibt es beim Einsatz dieser Notation einige Einschränkungen zu beachten, da der Unterstrich nicht an jeder beliebiger Stelle platziert werden kann, wie die im *nicht compilierbaren* Programm

```
1  public class UnterstrichFalsch {
2    public static void main(String[] args) {
3      // UNZULAESSIGE Verwendung des Unterstriches
4      float badPi1 = 3_.1415F;  // _ direkt vor dem Dezimalpunkt
```

```
5       float badPi2 = 3._1415F;   // _ direkt hinter dem Dezimalpunkt
6       long badNr = 99_99_99_L;   // _ direkt vor dem L-Anhang
7       int badNr1 = _42;          // _ am Anfang (waere ein Bezeichner)
8       int badNr2 = 42_;          // _ am Ende der Ziffernfolge
9       int badNr3 = 0_x42;        // _ innerhalb des 0x-Praefix
10      int badNr4 = 0x_42;        // _ am Anfang der Ziffernfolge
11      int badNr5 = 0x42_;        // _ am Ende
12      int badNr6 = 042_;         // _ am Ende
13    }
14  }
```

aufgeführten *unzulässigen* Beispiele demonstrieren.

3.3.2 Gleitkommatypen

Wir wollen eine einfache Rechnung durchführen. Das folgende Programm soll das Ergebnis von `1/10` ausgeben und verwendet dazu den Divisionsoperator:

```
1  public class Intdiv {
2    public static void main (String[] args) {
3      System.out.println(1/10);
4    }
5  }
```

Wir übersetzen das Programm und führen es aus. Zu unserer Überraschung erhalten wir jedoch ein vermeintlich falsches Ergebnis – und zwar die Null! Was ist geschehen?
Um zu begreifen, was eigentlich passiert ist, müssen wir uns eines klar machen: Wir haben mit ganzzahligen Datentypen gearbeitet. Der Divisionsoperator ist in Java jedoch so definiert, dass die Division zweier ganzer Zahlen wiederum eine ganze Zahl (nämlich den ganzzahligen Anteil des Quotienten) ergibt. Wir erinnern uns an die Grundschulzeit – hier hätte `1/10` ebenfalls 0 ergeben – mit Rest 1. Diesen Rest können wir in Java mit dem `%`-Zeichen bestimmen. Wir ändern unser Programm entsprechend:

```
1  public class Intdiv {
2    public static void main (String[] args) {
3      System.out.print("1/10 betraegt ");
4      System.out.print(1/10);  // ganzzahliger Anteil
5      System.out.print(" mit Rest ");
6      System.out.print(1%10);  // Rest
7    }
8  }
```

Das neue Programm gibt die folgende Meldung aus:

Konsole

```
1/10 betraegt 0 mit Rest 1
```

Nun ist es im Allgemeinen nicht wünschenswert, nur mit ganzen Zahlen zu arbeiten. Angenommen, wir wollen einen Geldbetrag in eine andere Währung umrechnen. Sollen die Pfennigbeträge dann etwa wegfallen? Java bietet aus diesem Grund auch die Möglichkeit, mit sogenannten **Gleitkommazahlen** (englisch: **floating point numbers**) zu arbeiten. Diese sind intern aus 32 bzw. 64 Bits aufgebaut,

wobei die Bitmuster jedoch anders interpretiert werden als bei den ganzzahligen Datentypen. Die in Java verfügbaren Gleitkommatypen **float** und **double** besitzen den in Tabelle 3.4 angegebenen Zahlenumfang.

Tabelle 3.4: Gleitkommatypen

Typname	größter positiver Wert	kleinster positiver Wert	Länge
float	≈3.4028234663852886E+038	≈1.4012984643248171E-045	32 Bits
double	≈1.7976931348623157E+308	≈4.9406564584124654E-324	64 Bits

Literalkonstanten für die Gleitkomma-Datentypen können aus verschiedenen optionalen Bestandteilen aufgebaut sein. Neben einem Dezimalpunkt können Ziffernfolgen (vor und nach dem Dezimalpunkt) und ein Exponent (bestehend aus einem `e` oder einem `E`, gefolgt von einer möglicherweise vorzeichenbehafteten Ziffernfolge) sowie eine Endung (`f`, `F`, `d` oder `D`) verwendet werden. Eine solche Gleitkommakonstante muss aber (zur Unterscheidung von ganzzahligen Konstanten) mindestens aus einem Dezimalpunkt oder einem Exponenten oder einer Endung bestehen. Falls ein Dezimalpunkt auftritt, muss vor oder nach ihm eine Ziffernfolge stehen.
Wie bereits erwähnt, steht hierbei das `E` mit anschließender Zahl X für die Multiplikation mit 10^X, d. h. die Zahl `1.2E3` steht für $1.2 \cdot 10^3$, also den Wert `1200`, die Zahl `1.2E-3` für $1.2 \cdot 10^{-3}$, also den Wert `0.0012`. Negative Zahlen werden erzeugt, indem man vor die entsprechende Zahl ein Minuszeichen setzt.
Ohne Endung oder mit der Endung `d` oder `D` ist eine Gleitkommakonstante stets vom Typ **double**. Verwenden wir die Endung `f` oder `F`, ist sie vom Typ **float**.
Natürlich kann mit 32 Bits oder auch 64 Bits nicht jede Zahl zwischen `+3.4028235E38` und `-3.4028235E38` exakt dargestellt werden. Der Computer arbeitet wieder mit Potenzen von 2, sodass selbst so einfache Zahlen wie `0.1` im Rechner nicht exakt codiert werden können. Es wird deshalb intern eine Rundung durchgeführt, sodass wir in einigen Fällen mit Rundungsfehlern rechnen müssen. Für den Moment soll uns das jedoch egal sein, denn wir wollen unser Programm nun auf das Rechnen mit Gleitkommazahlen umstellen. Hierzu ersetzen wir lediglich die ganzzahligen Werte durch Gleitkommazahlen:

```
1 public class Floatdiv {
2   public static void main (String[] args) {
3     System.out.print("1/10 betraegt ");
4     System.out.print(1.0/10.0);
5   }
6 }
```

Lassen wir das neue Programm laufen, so erhalten wir als Ergebnis:

Konsole

```
1/10 betraegt 0.1
```

3.3.3 Der Datentyp `char` für Zeichen

Manchmal erweist es sich als notwendig, nicht mit Zahlenwerten, sondern mit einzelnen Buchstaben oder Zeichen zu arbeiten. Diese Zeichen werden in Java durch den Datentyp **`char`** (Abkürzung für *character*) definiert. Literalkonstanten dieses Datentyps, d. h. einzelne Zeichen aus dem verfügbaren Zeichensatz, werden dabei in einfachen Hochkommata dargestellt, d. h. die Zeichen `a` und `?` hätten in Java die Darstellung `'a'` und `'?'`.
Daten vom Typ **`char`** werden intern mit 16 Bits (also 2 Bytes) dargestellt. Jedem Zeichen entspricht also intern eine gewisse Zahl oder auch Nummer (eben diese 16-Bit-Dualzahl), der sogenannte **Unicode**. Dem Buchstaben `a` entspricht beispielsweise die Nummer `97`. Man kann Werte vom Type **`char`** demnach auch als ganzzahlige Werte auffassen und entsprechend zu ganzzahligen Werten konvertieren. Beispielsweise erhält `i` durch die Anweisung `int i = 'a'` den Wert 97 zugewiesen (siehe auch Abschnitt 3.3.6).
Der Unicode-Zeichensatz enthält auch Zeichen, die möglicherweise in unserem Editor nicht dargestellt werden können. Um dennoch mit diesen Zeichen arbeiten zu können, stellt Java die **Unicode-Schreibweise** (`\u` gefolgt von vier hexadezimalen[10] Ziffern) zur Verfügung, mit der man alle Unicode-Zeichen (`\u0000` bis `\uffff`) darstellen kann. Dem Buchstaben `a` entspricht beispielsweise der Wert `\u0061` (der hexadezimale Wert 61 entspricht nämlich gerade dem dezimalen Wert 97), d. h. die Literalkonstante `'a'` kann als `'\u0061'` geschrieben werden.
Zur vereinfachten Darstellung von einigen „unsichtbaren" Zeichen und Zeichen mit vordefinierter Bedeutung als Kommando- oder Trennzeichen existiert außerdem die Notation mit sogenannten **Escape-Sequenzen**. Will man beispielsweise einen horizontalen Tabulator als Zeichenkonstante notieren, so kann man statt `'\u0009'` auch kurz `'\t'` schreiben. Ein Zeilenvorschub lässt sich als `'\n'` notieren. Die Symbole `'` und `"`, die zur Darstellung von Zeichen- oder Zeichenkettenkonstanten benötigt werden, können nur in der Form `'\''` und `'\"'` als Zeichenkonstante erzeugt werden. Als Konsequenz für die besondere Bedeutung des `\`-Zeichens muss die entsprechende Konstante in der Form `'\\'` notiert werden.

3.3.4 Zeichenketten

Mehrere Zeichen des Datentyps **`char`** können zu einer Zeichenkette (`String`) zusammengefasst werden. Solche Zeichenketten werden in Java allerdings nicht als Werte eines speziellen einfachen Datentyps, sondern als Objekte einer speziellen Klasse namens `String` behandelt (siehe Abschnitt 5.5.2). Darauf wollen wir jedoch hier noch nicht näher eingehen. Wichtig ist für uns lediglich, dass Literalkonstanten dieses Datentyps, d. h. eine Folge von Zeichen aus dem verfügbaren

[10] Sollten Sie bisher noch nicht mit hexadezimalen Ziffern in Berührung gekommen sein, können Sie im Glossar dieses Buches etwas darüber nachlesen. Aber keine Angst – für das Verstehen der nachfolgenden Abschnitte sind Kenntnisse über das Hexadezimalsystem nicht notwendig.

Zeichensatz, in doppelten Hochkommata dargestellt werden, d. h. die Zeichenkette `abcd` hätte in Java die Darstellung `"abcd"`.

3.3.5 Der Datentyp `boolean` für Wahrheitswerte

Häufig wird es notwendig sein, zwei Werte miteinander zu vergleichen. Ist etwa der Wert `23` größer, kleiner oder gleich einem anderen Wert? Die hierzu gegebenen Vergleichsoperatoren liefern eine Antwort, die letztendlich auf ja oder nein, wahr oder falsch, d. h. auf einen der Wahrheitswerte **`true`** oder **`false`** hinausläuft. Um mit solchen Wahrheitswerten arbeiten zu können, existiert in Java der Datentyp **`boolean`**. Dieser Typ besitzt lediglich zwei mögliche Werte: **`true`** und **`false`**. Dies sind somit auch die einzigen Literalkonstanten, die als **`boolean`**-Werte notiert werden können. Die Auswertung von logischen Ausdrücken (also beispielsweise von Vergleichen) liefert als Ergebnis Werte vom Typ **`boolean`**, die sich mit logischen Operatoren (siehe Abschnitt 3.4.2.4) weiter verknüpfen lassen, was beispielweise in der Ablaufsteuerung unserer Programme später eine wichtige Rolle spielen wird.

3.3.6 Implizite und explizite Typumwandlungen

Manchmal kommt es vor, dass wir an einer Stelle einen gewissen Datentyp benötigen, jedoch einen anderen vorliegen haben. Wir wollen etwa die Addition

```
9223372036854775000L + 807
```

einer 64-Bit-Zahl und einer 32-Bit-Zahl durchführen. Der Plus-Operator ist jedoch nur für Werte des gleichen Typs definiert. Was also tun?
In unserem Fall ist die Antwort einfach: Nichts. Der Java-Compiler erkennt, dass die linke Zahl einen Zahlenbereich hat, der den der rechten Zahl umfasst. Das System kann also die `807` problemlos in eine **`long`**-Zahl umwandeln (was es auch tut). Diese Umwandlung, die von uns unbemerkt im Hintergrund geschieht, wird als **implizite Typkonvertierung** (englisch: **implicite typecast**) bezeichnet.
Implizite Typkonvertierungen treten immer dann auf, wenn ein kleinerer Zahlenbereich in einen größeren abgebildet wird, d. h. von **`byte`** nach **`short`**, von **`short`** nach **`int`** und von **`int`** nach **`long`**. Ganzzahlige Datentypen lassen sich auch implizit in Gleitkommatypen umwandeln, obwohl hierbei eventuell Rundungsfehler auftreten können. Außerdem kann natürlich ein **`float`**-Wert automatisch nach **`double`** konvertiert werden. Eine implizite Umwandlung von **`char`** nach **`int`**, **`long`**, **`float`** oder **`double`** ist ebenfalls möglich.
Manchmal kommt es jedoch auch vor, dass wir einen größeren in einen kleineren Zahlenbereich umwandeln müssen. Wir haben beispielsweise das Ergebnis einer Gleitkommarechnung (sagen wir `3.14`) und interessieren uns nur für den Anteil vor dem Komma. Wir wollen also eine **`double`**-Zahl in einen **`int`**-Wert verwandeln.

Bei einer solchen Typumwandlung gehen eventuell Informationen verloren – der Compiler wird dies nicht ohne Weiteres tun. Er gibt uns beim Übersetzen eher eine Fehlermeldung der Form

Konsole

```
Incompatible type for declaration.
Explicit cast needed to convert double to int.
```

aus. Der Grund hierfür liegt darin, dass derartige Umwandlungen häufig auf Programmierfehlern beruhen, also eigentlich überhaupt nicht beabsichtigt sind. Der Compiler geht davon aus, dass ein Fehler vorliegt, und meldet dies auch.
Wir müssen dem Übersetzer also klarmachen, dass wir genau wissen, was wir tun! Dieses Verfahren wird als **explizite Typkonvertierung** (englisch: **explicite typecast**) bezeichnet und wird durchgeführt, indem wir den beabsichtigten Zieldatentyp in runden Klammern vor die entsprechende Zahl schreiben. In unserem obigen Beispiel würde dies etwa

```
(int) 3.14
```

bedeuten. Der Compiler erkennt, dass die Umwandlung wirklich gewollt ist, und schneidet die Nachkommastellen ab. Das Ergebnis der Umwandlung beträgt `3`.
Eine Umwandlung von **`boolean`** in einen anderen elementaren Datentyp oder umgekehrt ist nicht möglich – weder explizit noch implizit.
Achtung: Neben den bisher in diesem Abschnitt beschriebenen Situationen gibt es weitere Programmkontexte, in denen der Compiler automatische Typumwandlungen vornehmen kann. Insbesondere zu erwähnen ist dabei die Tatsache, dass bei Zuweisungen an Variablen vom Typ **`byte`**, **`short`** oder **`char`** der Wert rechts vom Zuweisungszeichen auch dann automatisch gewandelt werden kann, wenn es sich um einen konstanten Wert vom Typ **`int`** handelt (siehe auch Abschnitt 3.4.2.1) und dieser im Wertebereich der Variablen liegt. So können wir beispielsweise durch die Anweisung

```
short s = 1234;
```

der **`short`**-Variablen `s` den ganzzahligen Wert 1234 (also eigentlich eine 32-Bit-Zahl vom Typ **`int`**) zuweisen.
Weitere Konversionskontexte werden wir in den Kapiteln über Referenzdatentypen und Methoden kennenlernen.

3.3.7 Zusammenfassung

Wir haben einfache Datentypen kennengelernt, mit denen wir ganze Zahlen, Gleitkommazahlen, einzelne Zeichen und Wahrheitswerte in Java darstellen können. Wir haben erfahren, in welcher Beziehung diese Datentypen zueinander stehen und wie man sie ineinander umwandeln kann.

3.3.8 Übungsaufgaben

Aufgabe 3.6

Sie sollen verschiedene Variablen in einem Programm deklarieren. Finden Sie den passenden (möglichst speicherplatzsparenden) Typ für eine Variable, die angibt,

- wie viele Menschen in Deutschland leben,
- wie viele Menschen auf der Erde leben,
- ob es gerade Tag ist,
- wie hoch die Trefferquote eines Stürmers bei einem Fußballspiel ist,
- wie viele Semester Sie studieren werden,
- wie viele Studierende sich für einen Studiengang angemeldet haben,
- mit welchem Buchstaben Ihr Nachname beginnt.

Deklarieren Sie die Variablen und verwenden Sie sinnvolle Bezeichner.

Aufgabe 3.7

Welche der folgenden expliziten Typkonvertierungen ist unnötig, weil sie im Bedarfsfalle implizit durchgeführt würde?

a) `(int)3`

b) `(long)3`

c) `(long)3.1`

d) `(short)3`

e) `(short)3l`

f) `(double)3l`

g) `(int)'x'`

h) `(double)'x'`

Aufgabe 3.8

Welche impliziten Konvertierungen von ganzzahligen Werten in Gleitkommadatentypen können zu Rundungsfehlern führen? Geben Sie ein Beispiel an.

3.4 Der Umgang mit einfachen Datentypen

Wir haben nun die Wertebereiche einfacher Datentypen kennengelernt. Um damit arbeiten zu können, müssen wir lernen, wie sich Werte in Java speichern und durch Operatoren miteinander verknüpfen lassen.

3.4.1 Variablen

Bis jetzt waren unsere Beispielprogramme alle recht simpel. Dies lag vor allem daran, dass wir bislang keine Möglichkeit hatten, Werte zu speichern, um später wieder auf sie zugreifen zu können. Genau das erreicht man mit Variablen.
Am besten stellt man sich eine Variable wie ein Postfach vor: Ein Postfach ist im Prinzip nichts anderes als ein Behälter, der mit einem eindeutigen Schlüssel – in der Regel die Postfachnummer – gekennzeichnet ist und in den wir etwas hineinlegen können. Später können wir dann über den eindeutigen Schlüssel – die Postfachnummer – das Postfach wieder auffinden und auf den dort abgelegten Inhalt zugreifen. Allerdings sollte man sich stets der Tatsache bewusst sein, dass wir es mit ganz speziellen Postfächern zu tun haben, in denen niemals mehrere Briefe liegen können und die auch nur Briefe einer einzigen, genau auf die jeweiligen Fächer passenden Größe bzw. Form aufnehmen können.
Genauso verhält es sich nämlich mit Variablen: Eine Variable ist ein Speicherplatz. Über einen eindeutigen Schlüssel, in diesem Fall den Variablennamen, können wir auf eine Variable und damit auf den Speicherplatz zugreifen. Der Variablenname ist also eine Art symbolische **Adresse** (unsere Postfachnummer) für den Speicherplatz (unser Postfach). Man kann einer Variablen einen bestimmten Inhalt zuweisen und ihn später wieder auslesen. Das ist im Prinzip schon alles, was wir benötigen, um unsere Programme etwas interessanter zu gestalten. Abbildung 3.1 verdeutlicht diesen Zustand anhand der Variablen `b`, die den ganzzahligen Wert 107 beinhaltet.

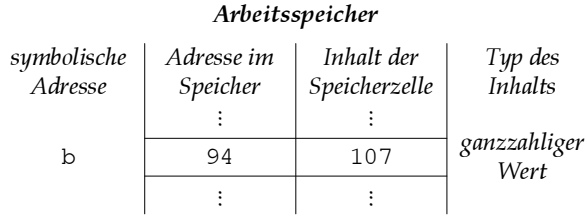

Abbildung 3.1: Einfaches schematisches Speicherbild

Betrachten wir ein einfaches Beispiel: Angenommen, wir wollten in einem Programm ausrechnen, wie viel Geld wir in unserem Nebenjob in der letzten Woche verdient haben. Unser Stundenlohn betrage in diesem Job 15 EUR, und letzte Woche haben wir insgesamt 18 Stunden gearbeitet. Dann könnten wir mit unserem bisherigen Wissen dazu ein Programm folgender Art basteln:

```
1 public class StundenRechner1 {
2   public static void main(String[] args) {
3     System.out.print("Arbeitsstunden: ");
4     System.out.println(18);
5     System.out.print("Stundenlohn in EUR: ");
6     System.out.println(15);
7     System.out.print("Damit habe ich letzte Woche ");
8     System.out.print(18 * 15);
```

```
9       System.out.println(" EUR verdient.");
10    }
11  }
```

Das Programm lässt sich natürlich anstandslos compilieren, und es erzeugt auch folgende (korrekte) Ausgabe:

Konsole

```
Arbeitsstunden: 18
Stundenlohn in EUR: 15
Damit habe ich letzte Woche 270 EUR verdient.
```

So weit, so gut. Was passiert aber nun in der darauffolgenden Woche, in der wir lediglich 12 Stunden Arbeitszeit absolvieren? Eine Möglichkeit wäre, einfach die Zahl der Arbeitsstunden im Programm zu ändern – allerdings müssten wir das jetzt an zwei Stellen tun, nämlich in den Zeilen 4 und 8 jeweils den Wert 18 auf 12 ändern. Dabei kann es nach drei (vier, fünf, ...) Wochen leicht passieren, dass wir vergessen, eine der beiden Zeilen zu ändern, oder dass wir uns in einer der Zeilen vertippen. Besser wäre es also, wenn wir die Anzahl der geleisteten Arbeitsstunden nur an einer einzigen Stelle im Programm ändern müssten. Gleiches gilt natürlich für den Arbeitslohn, der sich (hoffentlich) auch irgendwann einmal erhöht. Genau hier kommen nun Variablen ins Spiel.
Um Variablen in unserem Programm verwenden zu können, müssen wir dem Java-Compiler zunächst mitteilen, wie die Variablen heißen sollen und welche Art (also welchen Typ) von Werten wir in ihnen speichern wollen, sodass entsprechender Speicherplatz bereitgestellt werden kann. Diese Anweisung bezeichnen wir auch als **Deklaration**. Um in der Analogie der Postfächer zu bleiben: Wir müssen das Postfach mit einer bestimmten Größe (Brieffach, Paketfach, ...) erst einmal einrichten und es mit einer eindeutigen Postfachnummer versehen.
Eine solche Variablendeklaration hat stets folgende Form:

Syntaxregel

```
«VARIABLENTYP» «VARIABLENBEZEICHNER»;
```

Dabei entspricht «VARIABLENTYP» immer entweder einem einfachen Datentyp (**byte**, **short**, **int**, **long**, **float**, **double**, **char** oder **boolean**), einem Feldtyp oder einem Klassennamen (was die beiden letzten Varianten bedeuten, erfahren Sie später). «VARIABLENBEZEICHNER» ist eine eindeutige Zeichenfolge, die den in Abschnitt 3.1.2 beschriebenen Regeln für Bezeichner entspricht. Es hat sich eingebürgert, Variablennamen in Java in Kleinbuchstaben und ohne Sonderzeichen zusammenzuschreiben. Dabei wird mit einem Kleinbuchstaben begonnen und jedes neue Wort innerhalb des Bezeichners großgeschrieben, wie etwa in `tolleVariablenBezeichnung`. Daran wollen wir uns auch in Zukunft halten.
Um solchen Variablen nun Werte zuzuweisen, verwenden wir den **Zuweisungsoperator** =. Es geht aber dabei *nicht* um einen Vergleich, also um das Prüfen, ob der Wert auf der linken Seite des Gleichheitszeichens mit dem auf der rechten Seite angegebenen Wert identisch ist! Vielmehr wird durch den Zuweisungsoperator

der Wert des Ausdrucks auf der rechten Seite dem Speicherplatz der Variablen auf der linken Seite zugeordnet.
Natürlich können wir einer Variablen nur Werte zuweisen, die sich innerhalb des Wertebereichs des angegebenen Variablentyps befinden (siehe dazu auch Abschnitt 3.3). So könnten wir zum Beispiel, um eine Variable `a` vom Typ **`int`** zu deklarieren und ihr den Wert 5 zuzuweisen, Folgendes schreiben:

```
int a;
a = 5;
```

Wenn man nach einer Variablendeklaration gleich einen Wert in die Variable schreiben will, ist in Java auch folgende Kurzform erlaubt:

```
int a = 5;
```

Damit haben wir zwei Aufgaben auf einmal bewältigt, nämlich

1. die Deklaration, d. h. das Einrichten der Variablen, und
2. die **Initialisierung**, d. h. das Festlegen des ersten Wertes der Variablen.

Zurück zu unserem Beispiel. Wir wollten das Programm `StundenRechner1` so umschreiben, dass die Anzahl der geleisteten Arbeitsstunden nur noch an einer Stelle auftaucht. Die Lösung dafür liegt in der Verwendung einer Variablen für die Anzahl der Stunden. Das Programm sieht nun wie folgt aus:

```
1  public class StundenRechner2 {
2    public static void main(String[] args) {
3
4      int anzahlStunden = 12;
5      int stundenLohn   = 15;
6
7      System.out.print("Arbeitsstunden: ");
8      System.out.println(anzahlStunden);
9      System.out.print("Stundenlohn in EUR: ");
10     System.out.println(stundenLohn);
11     System.out.print("Damit habe ich letzte Woche ");
12     System.out.print(anzahlStunden * stundenLohn);
13     System.out.println(" EUR verdient.");
14
15   }
16 }
```

Die Zeilen 4 und 5 enthalten die benötigten Deklarationen und Initialisierungen der Variablen `anzahlStunden` und `stundenLohn`. In den Zeilen 8, 10 und 12 wird jetzt nur noch über den Variablennamen auf die Werte zugegriffen. Damit genügt, wenn wir nächste Woche unseren neuen Wochenlohn berechnen wollen, die Änderung einer einzigen Programmzeile.
Manchmal kann es sinnvoll sein, Variablen so zu vereinbaren, dass ihr Wert nach der Initialisierung im weiteren Programm nicht mehr verändert werden kann. Man spricht dann von sogenannten **`final`-Variablen** (oder auch von **symbolischen Konstanten**). Um eine solche unveränderliche Variable bzw. symbolische Konstante zu deklarieren, muss man der üblichen Deklaration mit Initialisierung das Schlüsselwort **`final`** voranstellen:

Syntaxregel

```
final «VARIABLENTYP» «VARIABLENBEZEICHNER» = «AUSDRUCK»;
```

Wollten wir also in unserem obigen Beispielprogramm dafür sorgen, dass unsere Variable `stundenLohn` zur symbolischen Konstante wird, so könnten wir sie einfach mit

```
final int STUNDEN_LOHN = 15;
```

deklarieren. Wir sind dabei der Konvention gefolgt, symbolische Konstanten stets mit Großbuchstaben zu schreiben, was wir auch in Zukunft tun werden. Jede nachfolgende Zuweisung an `STUNDEN_LOHN`, also z. B.

```
STUNDEN_LOHN = 152;
```

wäre demnach unzulässig.
Zum Schluss dieses Abschnitts noch eine Bemerkung der Vollständigkeit halber. Weiter hinten im Buch werden wir auch Datentypen anderer Art kennenlernen, die teilweise etwas längliche oder gar komplizierte Namen tragen. Im Zusammenhang mit der Deklaration und gleichzeitigen Initialisierung von Variablen solcher Datentypen kann manchmal eine abkürzende Notation sinnvoll sein. Diese Kurzschreibweise verzichtet auf die explizite Nennung des Datentyps und lässt sich prinzipiell auch für einfache Datentypen anwenden. Allerdings immer nur unter der Voraussetzung, dass der Compiler anhand der Initialisierung erkennen kann, um welchen Datentyp es sich handelt. In diesem Fall deklariert man eine Variable lediglich mit vorangestelltem **`var`**:

Syntaxregel

```
var «VARIABLENBEZEICHNER» = «AUSDRUCK»;
```

In unserem obigen Beispielprogramm könnten wir also die Variable `anzahlStunden` auch nur mit

```
var anzahlStunden = 15;
```

deklarieren sowie initialisieren, und der Compiler würde sie als **`int`**-Variable erkennen können. Wir werden in unserem Grundkursbuch aber auf diese Notation weitestgehend verzichten.

3.4.2 Operatoren und Ausdrücke

In der Regel will man mit Werten, die man in Variablen gespeichert hat, im Verlauf eines Programms mehr oder minder sinnvolle Berechnungen durchführen, die man im einfachsten Fall mit Hilfe komplexer Ausdrücke formulieren kann. Dazu stellt uns Java sogenannte Operatoren zur Verfügung. Auch in unserem letzten Beispielprogramm, `StundenRechner2`, haben wir schon verschiedene Operatoren benutzt, ohne näher darauf einzugehen. Das wollen wir jetzt nachholen.

Mit Operatoren lassen sich Werte, auch **Operanden** genannt, miteinander verknüpfen. Wie bereits beschrieben, kann man Operatoren nach der Anzahl ihrer Operanden in drei Kategorien einteilen. **Einstellige** Operatoren haben einen, **zweistellige** Operatoren zwei und **dreistellige** Operatoren drei Operanden. Synonym bezeichnet man diese Operatoren auch als **unär**, **binär** oder **ternär** bzw. **monadisch**, **dyadisch** oder **triadisch**.
Des Weiteren muss geklärt werden, in welcher Reihenfolge Operatoren und ihre Operanden in Java-Programmen geschrieben werden. Man spricht in diesem Zusammenhang auch von der **Notation** der Operatoren.
Die meisten einstelligen Operatoren werden in Java in der **Präfix**-Notation verwendet. Eine Ausnahme davon bilden die Inkrement- und Dekrementoperatoren, die sowohl in Präfix- als auch in **Postfix**-Notation verwendet werden können.[11]
Präfix-Notation bedeutet, dass der Operator vor seinem Operanden steht, also

Syntaxregel

```
«OPERATOR» «OPERAND»
```

Von **Postfix**-Notation spricht man hingegen, wenn der Operator hinter seinem Operanden steht, also

Syntaxregel

```
«OPERAND» «OPERATOR»
```

Zweistellige Operatoren in Java verwenden stets die **Infix**-Notation, in der der Operator zwischen seinen beiden Operanden steht, also

Syntaxregel

```
«OPERAND» «OPERATOR» «OPERAND»
```

Der einzige dreistellige Operator in Java, `?:` (siehe Abschnitt 3.4.2.4), benutzt ebenfalls die Infix-Notation, also

Syntaxregel

```
«OPERAND» ? «OPERAND» : «OPERAND»
```

Neben der Anzahl der Operanden kann man Operatoren auch nach dem **Typ** der Operanden einteilen.
Nach dieser (etwas längeren) Vorrede stellen wir in den folgenden Abschnitten die Operatoren von Java im Einzelnen vor, gruppiert nach dem Typ ihrer Operanden. Dabei müssen wir neben der Syntax der Ausdrücke, also deren korrekter Form, auch deren Semantik beschreiben, also die Bedeutung bzw. Wirkung der Operation auf den jeweiligen Daten angeben.

[11] Allerdings mit unterschiedlicher Bedeutung bzw. Semantik.

3.4.2.1 Arithmetische Operatoren

Arithmetische Operatoren sind Operatoren, die Zahlen, also Werte vom Typ **byte**, **short**, **int**, **long**, **float**, **double** oder **char**, als Operanden erwarten. Sie sind in den Tabellen 3.5 und 3.6 zusammengefasst.
Die Operatoren + und − können sowohl als zweistellige als auch als einstellige Operatoren gebraucht werden.

Tabelle 3.5: Zweistellige arithmetische Operatoren

Operator	Beispiel	Wirkung
+	`a + b`	Addiert `a` und `b`
−	`a - b`	Subtrahiert `b` von `a`
*	`a * b`	Multipliziert `a` und `b`
/	`a / b`	Dividiert `a` durch `b`
%	`a % b`	Liefert den Rest bei der ganzzahligen Division `a / b`

Tabelle 3.6: Einstellige arithmetische Operatoren

Operator	Beispiel	Funktion
+	`+ a`	Identität (liefert den gleichen Wert wie `a`)
−	`- a`	Negation (liefert den negativen Wert von `a`)

Achtung: Der Operator + kann auch dazu benutzt werden, um zwei Zeichenketten zu einer einzigen zusammenzufügen. So ergibt

```
"abcd" + "efgh"
```

die Zeichenkette

```
"abcdefgh"
```

Eine weitere Besonderheit stellt der **Ergebnistyp** arithmetischer Operationen dar. Damit meinen wir den Typ (also **byte**, **short**, **int**, **long**, **float**, **double**, **char** oder `String`) des Ergebnisses einer Operation, der durchaus nicht mit dem Typ beider Operanden übereinstimmen muss.
Bestes Beispiel dafür sind Programmzeilen wie etwa

```
short a = 1;
short b = 2;
short c = a + b;
```

die, obwohl dem Anschein nach korrekt, beim Compilieren zu folgender Fehlermeldung führen:

Konsole

```
Incompatible type for declaration.
Explicit cast needed to convert int to short.
```

Warum dies? Um den Ergebnistyp einer arithmetischen Operation zu bestimmen, geht der Java-Compiler wie folgt vor:

- Zunächst prüft er, ob einer der Operanden vom Typ **double** ist. Ist dies der Fall, so ist der Ergebnistyp dieser Operation **double**. Der andere Operand wird dann (falls notwendig) implizit nach **double** konvertiert und danach die Operation ausgeführt.
- War dies nicht der Fall, prüft der Compiler, ob einer der Operanden vom Typ **float** ist. Ist dies der Fall, so ist der Ergebnistyp dieser Operation **float**. Der andere Operand wird (falls erforderlich) dann implizit nach **float** konvertiert und danach die Operation ausgeführt.
- War dies auch nicht der Fall, so prüft der Compiler, ob einer der Operanden vom Typ **long** ist. Wenn ja, ist der Ergebnistyp dieser Operation **long**. Der andere Operand wird dann (falls notwendig) implizit nach **long** konvertiert und danach die Operation ausgeführt.
- Trat keiner der drei erstgenannten Fälle ein, so ist der Ergebnistyp dieser Operation auf jeden Fall **int**. Beide Operanden werden dann (falls erforderlich) implizit nach **int** konvertiert und danach die Operation ausgeführt.

Damit wird auch klar, warum obiges Beispiel eine Fehlermeldung produziert: Der Ausdruck `a + b` enthält keinen der Typen **double**, **float** oder **long**, daher wird der Ergebnistyp ein **int**. Diesen versuchen wir nun ohne explizite Typkonvertierung einer Variablen vom Typ **short** zuzuweisen, was zu einer Fehlermeldung führen muss, da der Wertebereich von **int** größer ist als der Wertebereich von **short**. Beheben lässt sich der Fehler jedoch ganz leicht, indem man explizit eine Typkonvertierung erzwingt. Die Zeilen

```
short a = 1;
short b = 2;
short c = (short)(a + b);
```

lassen sich daher anstandslos compilieren.
Was lernen wir daraus? Entweder verwenden wir ab jetzt für ganzzahlige Variablen nur noch den Typ **int** (hier tauchen diese Probleme nicht auf), oder aber wir achten bei jeder arithmetischen Operation darauf, das Ergebnis explizit in den geforderten Typ zu konvertieren. In jedem Falle aber wissen wir jetzt, wie wir Fehler dieser Art beheben können.
Achtung: Im Zusammenhang mit der Typwandlung wollen wir an dieser Stelle nochmals auf die Besonderheiten im Kontext der arithmetischen Operatoren hinweisen.

- Wird der Operator + dazu benutzt, einen Zeichenketten-Operanden (`String`-Operanden) und einen Operanden eines beliebigen anderen Typs zu verknüpfen, so wird der andere Operand implizit nach `String` gewandelt.
- Wie in Abschnitt 3.3.6 bereits erwähnt, kann der Wert eines arithmetischen Ausdrucks automatisch in den Typ **byte**, **short** oder **char** gewandelt werden, wenn es sich um einen konstanten Wert vom Typ **int** handelt. Man spricht in diesem Fall von einem **konstanten Ausdruck**, dessen Wert bereits beim Compilieren (also beim Übersetzen des Quelltexts in den Java-Bytecode) bestimmt werden kann.

Ganz allgemein darf ein konstanter Ausdruck lediglich Literalkonstanten und finale Variablen (symbolische Konstanten) der einfachen Datentypen oder Zeichenketten-Literale (`String`-Konstanten) enthalten. Zulässige konstante Ausdrücke wären also beispielweise

```
3 - 5.0 * 10          // Typ double
2 + 5 - 'a'           // Typ int
"good" + 4 + "you"    // Typ String
```

3.4.2.2 Bitoperatoren

Um diese Kategorie von Operatoren zu verstehen, müssen wir uns zunächst nochmals klarmachen, wie Werte im Computer gespeichert werden. Grundsätzlich kann ein Computer (bzw. die Elektronik, die in einem Computer enthalten ist) nur zwei Zustände unterscheiden: Aus oder An. Diesen Zuständen ordnen wir nun der Einfachheit halber die Zahlenwerte 0 und 1 zu. Die kleinste Speichereinheit, in der ein Computer genau einen dieser Werte speichern kann, nennen wir bekanntlich ein **Bit**. Um nun beliebige Zahlen und Buchstaben darstellen zu können, werden mehrere Bits zu neuen, größeren Einheiten zusammengefasst. Dabei entsprechen 8 Bits einem **Byte**, 1000 Bytes einem **Kilobyte** bzw. 1024 Bytes einem **Kibibyte** (siehe auch Abschnitt 1.1).
Bitoperatoren lassen ganzzahlige Operanden zu, arbeiten aber nicht mit dem ganzzahligen, eigentlichen Wert der Operanden, sondern nur mit deren Bits.[12] Auch hier unterscheidet man zwischen unären und binären Operationen. Die einzige unäre Operation, die **Negation** (dargestellt durch das Zeichen `~`), liefert bitweise stets das Komplement des Operanden, wie in Tabelle 3.7 dargestellt.

Tabelle 3.7: Bitweise Negation

a	~a
0	1
1	0

Daneben existieren drei binäre Operationen: das logische **Und** (dargestellt durch `&`), das logische **Oder** (`|`) und das logische **exklusive Oder** (`^`), deren bitweise Wirkungsweisen in Tabelle 3.8 dargestellt sind. Um also bei der Verknüpfung zweier Bits den Wert 1 zu erhalten, müssen

- bei der Operation `&` beide Bits den Wert 1 haben,
- bei der Operation `|` mindestens eines der Bits den Wert 1 haben und
- bei der Operation `^` genau eines der Bits den Wert 1 haben.

[12] Diese Operationen sind beispielsweise für die Definition und Manipulation selbst definierter Datentypen sinnvoll, bei denen bestimmte Dinge durch Bitketten einer bestimmten Länge codiert werden. Zur Bearbeitung dieser Bitketten benötigt man dann Operationen, die die einzelnen Bits in einer genau definierten Weise verändern.

Tabelle 3.8: Und, Oder und exklusives Oder

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bei der bitweisen Verknüpfung zweier ganzzahliger Operanden werden diese Bitoperationen auf mehrere Bits (Stelle für Stelle) gleichzeitig angewendet. So liefert beispielsweise das Programmstück

```
byte a = 9;
byte b = 3;
System.out.println(a & b);
System.out.println(a | b);
```

die Ausgabe

Konsole

```
1
11
```

weil der **byte**-Wert 9 dem Bitmuster `00001001` und der **byte**-Wert 3 dem Bitmuster `00000011` entspricht und somit die bitweise Verknüpfung `a & b` das Bitmuster `00000001`, also den dezimalen **byte**-Wert 1, liefert, während die bitweise Verknüpfung `a | b` das Bitmuster `00001011`, also den dezimalen **byte**-Wert 11 liefert. Die Java-Bitoperatoren sind in Tabelle 3.9 aufgelistet, während Tabelle 3.10 die sogenannten **Schiebeoperatoren** aufführt.

Tabelle 3.9: Bitoperatoren

Operator	Beispiel	Wirkung
`~`	`~ a`	Negiert `a` bitweise
`&`	`a & b`	Verknüpft `a` und `b` bitweise durch ein logisches Und
`|`	`a | b`	Verknüpft `a` und `b` bitweise durch ein logisches Oder
`^`	`a ^ b`	Verknüpft `a` und `b` bitweise durch ein logisches exklusives Oder

Tabelle 3.10: Schiebeoperatoren

Operator	Beispiel	Funktion
`<<`	`a << b`	Schiebt die Bits in `a` um `b` Stellen nach links und füllt mit 0-Bits auf
`>>`	`a >> b`	Schiebt die Bits in `a` um `b` Stellen nach rechts und füllt mit dem höchsten Bit von `a` auf
`>>>`	`a >>> b`	Schiebt die Bits in `a` um `b` Stellen nach rechts und füllt mit 0-Bits auf

Schiebeoperatoren verschieben alle Bits eines ganzzahligen Wertes um eine vorgegebene Anzahl von Stellen nach links bzw. rechts. Das Schieben der Bits um eine Stelle nach links bzw. rechts kann auch als Multiplikation bzw. als Division des Wertes mit bzw. durch 2 interpretiert werden.

3.4.2.3 Zuweisungsoperator

Eine Sonderstellung unter den Operatoren nimmt der **Zuweisungsoperator** = ein. Mit ihm kann man einer Variablen Werte zuordnen. Der Ausdruck

```
a = 3;
```

ordnet beispielsweise der Variablen `a` den Wert `3` zu. Rechts vom Zuweisungszeichen kann nicht nur ein konstanter Wert, sondern auch eine Variable oder ein Ausdruck stehen. Um den gleichen Wert mehreren Variablen gleichzeitig zuzuordnen, kann man auch ganze Zuordnungsketten bilden, etwa

```
a = b = c = 5;
```

Hier wird der Wert `5` allen drei Variablen `a`, `b`, `c` zugeordnet. Möglich wird dies deshalb, weil jede Zuweisung selbst wieder ein Ausdruck ist, dessen Wert der Wert der linken Seite ist, der wiederum an die jeweils nächste linke Seite weitergegeben werden kann.
Achtung: Der Zuweisungsoperator = hat grundsätzlich nichts mit der aus der Mathematik bekannten Gleichheitsrelation (Identität) zu tun, die das gleiche Zeichen = benutzt. Ein mathematischer Ausdruck der Form

```
a = a + 1
```

ist für eine reelle Zahl `a` natürlich falsch, die Java-Anweisung

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

dagegen ist syntaktisch völlig korrekt und erhöht den Wert der Variablen `a` um 1. Will man mit dem Wert einer Variablen Berechnungen anstellen und das Ergebnis danach in der gleichen Variablen speichern, ist es oft lästig, den Variablennamen sowohl links als auch rechts des Zuweisungsoperators zu tippen. Daher bietet Java für viele binäre Operatoren auch eine verkürzende Schreibweise an. Statt

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

kann man daher auch kürzer

```
a += 1;
```

schreiben. Beide Ausdrücke sind in Java völlig äquivalent, beide erhöhen den Wert der Variablen `a` um 1. Tabelle 3.11 fasst alle möglichen abkürzenden Schreibweisen zusammen.

3.4.2.4 Vergleichsoperatoren und logische Operatoren

Eine weitere Gruppe von Operatoren bilden die sogenannten **Vergleichsoperatoren**. Diese stets binären Operatoren vergleichen ihre Operanden miteinander und geben immer ein Ergebnis vom Typ **`boolean`**, also entweder **`true`** oder **`false`**, zurück. Im Einzelnen sind dies die in Tabelle 3.12 aufgeführten Operatoren.

Tabelle 3.11: Abkürzende Schreibweisen für binäre Operatoren

Abkürzung	Beispiel	äquivalent zu
`+=`	`a += b`	`a = a + b`
`-=`	`a -= b`	`a = a - b`
`*=`	`a *= b`	`a = a * b`
`/=`	`a /= b`	`a = a / b`
`%=`	`a %= b`	`a = a % b`
`&=`	`a &= b`	`a = a & b`
`|=`	`a |= b`	`a = a | b`
`^=`	`a ^= b`	`a = a ^ b`
`<<=`	`a <<= b`	`a = a << b`
`>>=`	`a >>= b`	`a = a >> b`
`>>>=`	`a >>>= b`	`a = a >>> b`

Tabelle 3.12: Vergleichsoperatoren

Operator	Beispiel	liefert genau dann **`true`**, wenn ...
`>`	`a > b`	... `a` größer als `b` ist
`>=`	`a >= b`	... `a` größer als oder gleich `b` ist
`<`	`a < b`	... `a` kleiner als `b` ist
`<=`	`a <= b`	... `a` kleiner als oder gleich `b` ist
`==`	`a == b`	... `a` gleich `b` ist
`!=`	`a != b`	... `a` ungleich `b` ist

Um nun komplexe Ausdrücke zu erstellen, werden die Vergleichsoperatoren meist durch sogenannte **logische Operatoren** verknüpft. Diese ähneln auf den ersten Blick den schon vorgestellten Bitoperatoren. Allerdings erwarten logische Operatoren stets Operanden vom Typ **`boolean`**, und ihr Ergebnistyp ist ebenfalls **`boolean`**. Wie bei den Bitoperatoren existieren auch hier Operatoren für logisches Und (Operator `&`), logisches Oder (Operatoren `|` und `^`) und Negation (Operator `!`), deren Wirkung Tabelle 3.13 darstellt.

Tabelle 3.13: Logisches Und, Oder und die Negation

a	b	a & b	a | b	a ^ b	! a
`false`	**`false`**	**`false`**	**`false`**	**`false`**	**`true`**
`false`	**`true`**	**`false`**	**`true`**	**`true`**	**`true`**
`true`	**`false`**	**`false`**	**`true`**	**`true`**	**`false`**
`true`	**`true`**	**`true`**	**`true`**	**`false`**	**`false`**

Eine Besonderheit stellen die Operatoren `&&` und `||` dar. Bei ihnen wird der zweite Operand nur dann ausgewertet, wenn das Ergebnis der Operation nicht schon nach Auswertung des ersten Operanden klar ist. Im Fall `a && b` muss also `b` nur dann ausgewertet werden, wenn die Auswertung von `a` den Wert **`true`** ergibt. Im Fall `a || b` muss `b` nur dann ausgewertet werden, wenn `a` den Wert **`false`** ergibt.

Ist also beispielsweise der Ausdruck `(a > 15) && (b < 20)` zu bewerten und der Wert der Variablen `a` gerade `10`, so ergibt der erste Teilausdruck `(a > 15)` zunächst **`false`**. Der Compiler prüft in diesem Fall den Wert der Variablen `b` gar nicht mehr nach, da ja der gesamte Ausdruck auch nur noch **`false`** sein kann.
Was haben wir nun davon? Zunächst kann man durch geschickte Ausnutzung dieser Operatoren im Einzelfall die Ausführungsgeschwindigkeit des Programms deutlich steigern. Müssen an einer Stelle eines Programms zwei Bedingungen auf **`true`** überprüft werden und ist das Ergebnis der einen Bedingung in 90% aller Fälle **`false`**, so empfiehlt es sich, diese Bedingung zuerst überprüfen zu lassen und die zweite über den bedingten Operator anzuschließen. Jetzt muss die zweite Bedingung nur noch in den 10% aller Fälle überprüft werden, in denen die erste Bedingung wahr wird. In allen anderen Fällen läuft das Programm schneller ab. Des Weiteren lässt sich, falls die Bedingungen nicht nur Variablen, sondern auch Aufrufe von Methoden enthalten (was das genau ist, erfahren wir später), mit Hilfe eines bedingten Operators erreichen, dass bestimmte Programmteile überhaupt nicht abgearbeitet werden. Doch dazu später mehr.
Tabelle 3.14 fasst alle logischen Operatoren zusammen.

Tabelle 3.14: Logische Operatoren

Operator	Beispiel	Funktion
`&`	`a & b`	Verknüpft `a` und `b` durch ein logisches Und
`&&`	`a && b`	Verknüpft `a` und `b` durch ein logisches Und (nur bedingte Auswertung von `b`)
`|`	`a | b`	Verknüpft `a` und `b` durch ein logisches Oder
`||`	`a || b`	Verknüpft `a` und `b` durch ein logisches Oder (nur bedingte Auswertung von `b`)
`^`	`a ^ b`	Verknüpft `a` und `b` durch ein logisches exklusives Oder
`!`	`! a`	Negiert `a`

Eine Sonderstellung unter den Vergleichs- und logischen Operatoren nimmt der dreistellige Bedingungsoperator **`?:`** ein. Eigentlich stellt er nur eine verkürzende Schreibweise für eine **`if`**-Entscheidungsanweisung dar (siehe Abschnitt 3.5.3). Als ersten Operanden erwartet er einen Ausdruck mit Ergebnistyp **`boolean`**, als zweiten und dritten jeweils Ausdrücke, die beide von einem numerischen Datentyp, beide vom Typ **`boolean`** oder beide vom Typ `String` sind.[13] Liefert der erste Operand **`true`** zurück, so gibt der Operator den Wert seines zweiten Operanden zurück. Liefert der erste Operand **`false`**, so ist der Wert des dritten Operanden das Ergebnis der Operation.
Beispiel: Der Ausdruck

```
(a == 15) ? "a ist 15" : "a ist nicht 15"
```

[13] Genauer gesagt können beide auch von einem beliebigen anderen Referenzdatentyp sein. Auf solche Datentypen gehen wir jedoch erst in Kapitel 4 ein.

liefert die Zeichenkette `a ist 15`, falls die Variable `a` den Wert `15` enthält, und die Zeichenkette `a ist nicht 15` in allen anderen Fällen.

3.4.2.5 Inkrement- und Dekrementoperatoren

Auch hier handelt es sich eigentlich nur um verkürzte Schreibweisen von häufig verwendeten Ausdrücken. Um den Inhalt der Variablen `a` um eins zu erhöhen, könnten wir – wie wir mittlerweile wissen – beispielsweise schreiben:

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

Alternativ bietet sich – auch das haben wir schon gelernt – der verkürzte Zuweisungsoperator an, d. h.

```
a += 1;
```

In diesem speziellen Fall (Erhöhung des Variableninhaltes um genau `1`) bietet sich jetzt eine noch kürzere Schreibweise an, nämlich

```
a++;
```

Der **Inkrementoperator** `++` ist also unär und erhöht den Wert seines Operanden um eins. Analog dazu erniedrigt der **Dekrementoperator** `--`, ebenfalls ein unärer Operator, den Wert seines Operanden um eins. Beide Operatoren dürfen nur auf Variablen angewendet werden.
Was bleibt zu beachten? Wie bereits erwähnt, können beide Operatoren sowohl in Präfix- als auch in Postfix-Notation verwendet werden. Wird der Operator in einer isolierten Anweisung – wie in obigem Beispiel – verwendet, so sind beide Notationen äquivalent.
Sind Inkrement- bzw. Dekrementoperator jedoch Teil eines größeren Ausdrucks, so hat die Notation entscheidenden Einfluss auf das Ergebnis des Ausdrucks. Bei Verwendung der Präfix-Notation wird der Wert der Variablen *erst* erhöht bzw. erniedrigt und *dann* der Ausdruck ausgewertet. Analog dazu wird bei der Postfix-Notation *zuerst* der Ausdruck ausgewertet und *dann* erst das Inkrement bzw. Dekrement durchgeführt.
Beispiel:

```
a = 5;
b = a++;
c = 5;
d = --c;
```

Nach Ausführung dieses Programmsegments enthält `b` den Wert `5`, da *zuerst* der Ausdruck ausgewertet und *dann* das Inkrement ausgeführt wird. `d` dagegen enthält den Wert `4`, da hier *zuerst* das Dekrement durchgeführt und *dann* der Ausdruck ausgewertet wird. Am Ende haben `a` den Wert 6 und `c` den Wert 4.

3.4.2.6 Priorität und Auswertungsreihenfolge der Operatoren

Bislang haben wir alle Operatoren nur isoliert betrachtet, d. h. unsere Ausdrücke enthielten jeweils nur einen Operator. Verwendet man jedoch mehrere Operatoren in einem Ausdruck, stellt sich die Frage, in welcher Reihenfolge die einzelnen

Operationen ausgeführt werden. Dies ist durch die Prioritäten der einzelnen Operatoren festgelegt. Dabei werden Operationen höherer Priorität stets vor Operationen niedrigerer Priorität ausgeführt, wenn nicht Klammern (siehe unten) dies anders regeln. Haben mehrere zweistellige Operationen, die im gleichen Ausdruck stehen, die gleiche Priorität, so wird – außer bei Zuweisungsoperatoren – stets von links nach rechts ausgewertet. Der Zuweisungsoperator = sowie alle verkürzten Zuweisungsoperatoren – also +=, −=, usw. – werden, wenn sie nebeneinander in einem Ausdruck vorkommen, dagegen von rechts nach links ausgewertet.
Tabelle 3.15 enthält alle Operatoren, die wir bisher kennengelernt haben, geordnet nach deren Priorität. Mit der obersten Gruppe von Operatoren mit höchster Priorität (15) werden wir uns erst in den Kapiteln über Referenzdatentypen (Felder und Klassen) und über Methoden näher beschäftigen.

Tabelle 3.15: Priorität der Operatoren

Bezeichnung	Operator	Priorität
Komponentenzugriff bei Klassen	`.`	15
Komponentenzugriff bei Feldern	`[ ]`	15
Methodenaufruf	`( )`	15
Unäre Operatoren	`++, –, +, –, ~, !`	14
Explizite Typkonvertierung	`( )`	13
Multiplikative Operatoren	`*, /, %`	12
Additive Operatoren	`+, –`	11
Schiebeoperatoren	`<<, >>, >>>`	10
Vergleichsoperatoren	`<, >, <=, >=`	9
Vergleichsoperatoren (Gleichheit/Ungleichheit)	`==, !=`	8
bitweises bzw. logisches Und	`&`	7
bitweises exklusives Oder	`^`	6
bitweises bzw. logisches Oder	`|`	5
logisches Und	`&&`	4
logisches Oder	`||`	3
Bedingungsoperator	`?:`	2
Zuweisungsoperatoren	`=, +=, −=` usw.	1

Praktisch bedeutet dies für uns, dass wir bedenkenlos „Punkt-vor-Strich-Rechnung“ verwenden können, ohne uns über Prioritäten Gedanken machen zu müssen. Da multiplikative Operatoren eine höhere Priorität als additive haben, werden Ausdrücke wie z. B.

```
4 + 3 * 2
```

wie erwartet korrekt ausgewertet (hier: Ergebnis ist `10`, nicht `14`). Darüber hinaus sollten wir aber lieber ein Klammernpaar zu viel als zu wenig verwenden, um die gewünschte Ausführungsreihenfolge der Operationen zu garantieren. Mit den runden Klammern `()` können wir nämlich, genau wie in der Mathematik, die

Reihenfolge der Operationen eindeutig festlegen, gleichgültig, welche Priorität ihnen zugeordnet ist.

3.4.3 Allgemeine Ausdrücke

Wie wir gesehen haben, setzen sich Ausdrücke in Java aus Operatoren und Operanden zusammen. Die Operanden selbst können dabei wieder

- Konstanten,
- Variablen,
- geklammerte Ausdrücke oder
- Methodenaufrufe

sein. Die letztgenannten Methodenaufrufe werden wir erst später genauer kennenlernen. Wir wollen daher im Folgenden nur etwas Ähnliches wie Aufrufe von mathematischen Standardfunktionen (wie z. B. sin oder cos) darunter verstehen. In Java sind diese Funktionen, wie bereits erwähnt, nicht im Sprachkern enthalten. Sie wurden ausgelagert in die Klasse `Math`. Wir können sie daher nur mit dem vorgestellten Klassennamen (also z. B. `Math.sin(5.3)`) aufrufen.
Sind beide Operanden einer Operation selbst wieder Ausdrücke (also z. B. geklammerte Operationen oder Methodenaufrufe), wird immer erst der linke und dann der rechte Operand berechnet. Der Ausdruck in der Java-Programmzeile

```
b = Math.sqrt(3.5 + x) * 5 / 3 - (x + 10) * (x -4.1) < 0;
```

wird somit gemäß den Prioritäten wie folgt abgearbeitet (die Zwischenergebnisse haben wir der Einfachheit halber mit `z1` bis `z8` durchnummeriert):

```
z1 = 3.5 + x;
z2 = Math.sqrt(z1);
z3 = z2 * 5;
z4 = z3 / 3;
z5 = x + 10;
z6 = x - 4.1;
z7 = z5 * z6;
z8 = z4 - z7;
 b = z8 < 0;
```

3.4.4 Ein- und Ausgabe

Da wir nun mit den einfachen Datentypen umgehen können, wird sich natürlich auch die Notwendigkeit ergeben, Werte für Ausdrücke dieser Datentypen auf die Konsole auszugeben bzw. Werte für Variablen dieser Datentypen einzulesen. Wie bereits erwähnt, stellen wir für Letzteres die Klasse `IOTools` zur Verfügung, in der für jede Art von einzulesendem Wert (ganze Zahl, Gleitkommawert, logischer Wert etc.) eine entsprechende Methode bereitgestellt wird. Diese Methoden sind im Anhang C detailliert beschrieben. Wir wollen uns hier zumindest noch ein kleines Beispielprogramm anschauen, in dem einige dieser Methoden verwendet

werden und das gleichzeitig die Verwendung der `println`-Methode verdeutlicht.

```
1  import Prog1Tools.IOTools;
2
3  public class IOToolsTest {
4
5    public static void main (String[] args) {
6
7      int     i, j, k;
8      double  d;
9      char    c;
10     boolean b;
11
12     // int-Eingabe ohne Prompt (ohne vorherige Ausgabe)
13     i = IOTools.readInteger();
14
15     // int-Eingabe mit Prompt
16     System.out.print("j = ");
17     j = IOTools.readInteger();
18
19     // Vereinfachte int-Eingabe mit Prompt
20     k = IOTools.readInteger("k = ");
21
22     // double-Eingabe mit Prompt
23     d = IOTools.readDouble("d = ");
24
25     // char-Eingabe mit Prompt
26     c = IOTools.readChar("c = ");
27
28     // boolean-Eingabe mit Prompt
29     b = IOTools.readBoolean("b = ");
30
31     // Testausgaben
32     System.out.println("i = " + i);
33     System.out.println("j = " + j);
34     System.out.println("k = " + k);
35     System.out.println("d = " + d);
36     System.out.println("c = " + c);
37     System.out.println("b = " + b);
38   }
39 }
```

Wenn wir dieses Programm übersetzen und starten, könnte sich (natürlich abhängig von unseren Benutzereingaben) folgender Programmablauf ergeben (zur Verdeutlichung unserer Eingaben haben wir diese etwas nach rechts verschoben; in der linken Spalte ist also jeweils zu sehen, was das Programm ausgibt, in der rechten Spalte stehen unsere Eingaben):

Konsole

```
                                                    123
j =                                                 a
Eingabefehler
java.lang.NumberFormatException: a
Bitte Eingabe wiederholen...
```

```
                                                    1234
k =                                                 12345
d =                                                 123.456789
c =                                                 x
b =                                                 true
i = 123
j = 1234
k = 12345
d = 123.456789
c = x
b = true
```

Insbesondere bei der ersten Eingabe erkennen wir, wie wichtig es ist, vor jeder Eingabe zumindest eine kurze Information darüber auszugeben, dass nun eine Eingabe erfolgen soll. Man spricht auch von einem sogenannten **Prompt** (deutsch: Aufforderung). Ohne diese Ausgabe (wie beim ersten Eingabebeispiel) scheint das Programm nämlich erst mal zu „hängen“, weil wir nicht sofort merken, dass wir schon etwas eingeben können.

3.4.4.1 Statischer Import der IOTools-Methoden

Die Anwendung der `IOTools`-Methoden lässt sich deutlich vereinfachen, da es möglich ist, die statischen Methoden einer Klasse, z. B. eben der Klasse `IOTools`, so zu importieren, dass sie ohne den vorangestellten Klassennamen verwendet werden können.
Der statische Import einer einzelnen Methode wird syntaktisch in der Form

Syntaxregel

```
import static «PAKETNAME».«KLASSENNAME».«METHODENNAME»;
```

angegeben. Sollen alle Klassenmethoden einer Klasse importiert werden, so wird dies durch

Syntaxregel

```
import static «PAKETNAME».«KLASSENNAME».*;
```

angezeigt.
Nachfolgend nun eine Version unseres weiter oben angegebenen Programms `IOToolsTest`, in der wir alle Methoden der Klasse `IOTools` statisch importieren. Die Aufrufe der Einlesemethoden fallen nun deutlich kürzer aus.

```
1  import static Prog1Tools.IOTools.*;
2
3  public class IOToolsTestMitStaticImport {
4    public static void main (String[] args) {
5      int     i, j, k;
```

```
6       double  d;
7       char    c;
8       boolean b;
9
10      // int-Eingabe ohne Prompt (ohne vorherige Ausgabe)
11      i = readInteger();
12
13      // int-Eingabe mit Prompt
14      System.out.print("j = ");
15      j = readInteger();
16
17      // Vereinfachte int-Eingabe mit Prompt
18      k = readInteger("k = ");
19
20      // double-Eingabe mit Prompt
21      d = readDouble("d = ");
22
23      // char-Eingabe mit Prompt
24      c = readChar("c = ");
25
26      // boolean-Eingabe mit Prompt
27      b = readBoolean("b = ");
28
29      // Testausgaben
30      System.out.println("i = " + i);
31      System.out.println("j = " + j);
32      System.out.println("k = " + k);
33      System.out.println("d = " + d);
34      System.out.println("c = " + c);
35      System.out.println("b = " + b);
36    }
37  }
```

3.4.5 Zusammenfassung

Im letzten, zugegebenermaßen etwas längeren, Abschnitt haben wir gelernt, was Variablen sind (Analogie: Postfächer), haben Operatoren kennengelernt, mit denen wir Werte verknüpfen und zu Ausdrücken kombinieren können.

3.4.6 Übungsaufgaben

Aufgabe 3.9

Die nachfolgenden Programmfragmente weisen jeweils einen syntaktischen bzw. semantischen Fehler auf und lassen sich daher nicht compilieren. Finden Sie die Fehler, und begründen Sie kurz Ihre Wahl.

a) `boolean false, println;`

b) `char ab, uv, x y;`

c) `int a = 0x1, c = 1e2;`

d) `double a, b_c, d-e;`

e) `int mo, di, mi, do, fr, sa, so;`

f) `System.out.println("10 = ", 10);`

Aufgabe 3.10

Schreiben Sie ein Programm, das Sie auffordert, Name und Alter einzugeben. Das Programm soll Sie danach mit Ihrem Namen begrüßen und Ihr Alter *in Tagen* ausgeben. Verwenden Sie die `IOTools`.
Hinweis: Für die Umwandlung des Alters in Tage brauchen Sie die Schaltjahre nicht zu berücksichtigen.

Aufgabe 3.11

Gegeben sei das folgende Java-Programm:

```
1  public class Plus {
2    public static void main (String args []) {
3      int a = 1, b = 2, c = 3, d = 4;
4      System.out.println(++a);
5      System.out.println(a);
6      System.out.println(b++);
7      System.out.println(b);
8      System.out.println((++c) + (++c));
9      System.out.println(c);
10     System.out.println((d++) + (d++));
11     System.out.println(d);
12   }
13 }
```

Vollziehen Sie das Programm nach, und überlegen Sie sich, welche Werte ausgegeben werden.

Aufgabe 3.12

Bei der Ausgabe mehrerer Werte mit nur einer `System.out.print`- oder `System.out.println`-Anweisung müssen die auszugebenden Werte mittels + als Strings (Zeichenketten) miteinander verknüpft werden.

a) Warum kann man die auszugebenden Werte nicht einfach als Kommaliste aufzählen?

b) Was passiert, wenn man einen `String`-Operanden mit einem Operanden eines beliebigen anderen Datentyps mittels + verknüpft?

c) Stellen Sie bei den nachfolgenden Ausgabeanweisungen fest, welche zulässig und welche aufgrund eines fehlerhaften Ausdrucks im Argument der `println`-Methode unzulässig sind.

Korrigieren Sie die unzulässigen Anweisungen, indem Sie eine geschickte Klammerung einbauen. Geben Sie an, was ausgegeben wird.

```
double x = 1.0, y = 2.5;
System.out.println(x / y);
System.out.println("x / y = " + x / y);
System.out.println(x + y);
System.out.println("x + y = " + x + y);
System.out.println(x - y);
System.out.println("x - y = " + x - y);
System.out.println(1 + 2 + 3 + 4);
System.out.println(1 + 2 + 3 + "4");
System.out.println("1" + 2 + 3 + 4);
System.out.println("Hilfe" + true + 3);
System.out.println(true + 3 + "Hilfe");
```

Aufgabe 3.13

Ziel dieser Aufgabe ist es, die Formulierung von arithmetischen Ausdrücken in der Syntax der Programmiersprache Java zu üben. Doch Vorsicht: Bei der Auswertung von arithmetischen Ausdrücken auf einer Rechenanlage muss das berechnete Ergebnis nicht immer etwas mit dem tatsächlichen Wert des Ausdrucks zu tun haben. Denn die Auswertung ist bedingt durch die endliche Zahlendarstellung auf dem Rechner stets Rundungsfehlern ausgesetzt, die sich unter Umständen zu gravierenden Fehlern akkumulieren können. Was tatsächlich passieren kann, können Sie nach Bearbeiten dieser Aufgabe ermessen.
Schreiben Sie ein Java-Programm, das unter der Verwendung von Variablen vom Typ **`double`** bestimmte Ausdruckswerte berechnet und deren Ergebnis auf dem Bildschirm ausgibt.

a) Berechnen Sie den Wert

$$x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4 + x_5y_5 + x_6y_6$$

für $x_1 = 10^{20}$, $x_2 = 1223$, $x_3 = 10^{18}$, $x_4 = 10^{15}$, $x_5 = 3$, $x_6 = -10^{12}$ und für $y_1 = 10^{20}$, $y_2 = 2$, $y_3 = -10^{22}$, $y_4 = 10^{13}$, $y_5 = 2111$, $y_6 = 10^{16}$.

Das *richtige* Ergebnis ist übrigens 8779.

b) Berechnen Sie den Wert

$$\frac{1}{107751}(1682xy^4 + 3x^3 + 29xy^2 - 2x^5 + 832)$$

für $x = 192119201$ und $y = 35675640$. Verwenden Sie dabei nur die Grundoperationen $+$, $-$, $*$ und $/$, und stellen Sie Ausdrücke wie x^2 bzw. x^4 als $x * x$ bzw. $x^2 * x^2$ dar.

c) Durch eine algebraische Umformung lässt sich eine äquivalente Darstellung für diesen zweiten Ausdruck finden, z. B.

$$\frac{xy^2}{107751}(1682y^2 + 29) + \frac{x^3}{107751}(3 - 2x^2) + \frac{832}{107751}.$$

Vergleichen Sie das Ergebnis für die Auswertung dieser Darstellung mit dem zuvor berechneten. Können Sie abschätzen, welches Ergebnis richtig ist, falls dies überhaupt für eines zutrifft?

Der *richtige* Wert des Ausdrucks ist 1783.

Aufgabe 3.14

Stellen Sie sich vor, Sie machen gerade Urlaubsvertretung für einen Verpackungsingenieur bei der Firma *Raviolita*. Dieser hat Ihnen kurz vor seiner Abreise in den Spontanurlaub noch das Programm (bzw. die Klasse) `Raviolita` hinterlassen:

```
1  public class Raviolita {
2    public static void main (String[] args) {
3      final double PI = 3.141592;
4      double u, h;
5      u =      ;  // geeignete Testwerte einbauen
6      h =      ;  // geeignete Testwerte einbauen
7
8      // nachfolgend die fehlenden Deklarationen ergaenzen
9
10     // nachfolgend die fehlenden Berechnungen ergaenzen
11
12     // nachfolgend die fehlenden Ausgaben ergaenzen
13   }
14 }
```

Dieses Programm führt Berechnungen durch, die bei der Herstellung von Konservendosen aus einem Blechstück mit

- Länge u (Umfang der Dose in Zentimetern) und
- Breite h (Höhe der Dose in Zentimetern)

anfallen. Dieses Programm sollen Sie nun so vervollständigen, dass es ausgehend von den Variablen u und h und unter Verwendung der Konstanten π (bzw. `PI = 3.141592`) die folgenden Werte berechnet und ausgibt:

- den Durchmesser des Dosenbodens: $d_{boden} = \frac{u}{\pi}$,
- die Fläche des Dosenbodens: $f_{boden} = \pi \cdot (\frac{d_{boden}}{2})^2$,
- die Mantelfläche der Dose: $f_{mantel} = u \cdot h$,
- die Gesamtfläche der Dose: $f_{gesamt} = 2 \cdot f_{boden} + f_{mantel}$,
- das Volumen der Dose: $v = f_{boden} \cdot h$.

Testen Sie Ihr Programm mit vernünftigen Daten für u und h.

Aufgabe 3.15

Schreiben Sie ein Java-Programm, das eine vorgegebene Zahl von Sekunden in Jahre, Tage, Stunden, Minuten und Sekunden zerlegt.
Das Programm soll z. B. für einen Sekundenwert 158036522 Folgendes ausgeben:

Konsole

```
158036522 Sekunden entsprechen:
5 Jahren,
4 Tagen,
3 Stunden,
2 Minuten und
2 Sekunden.
```


3.5 Anweisungen und Ablaufsteuerung

Als letzte Grundelemente der Sprache Java lernen wir in den folgenden Abschnitten Befehle kennen, mit denen wir den Ablauf unseres Programms beeinflussen, d. h. bestimmen können, ob und in welcher Reihenfolge bestimmte Anweisungen unseres Programms ausgeführt werden. In diesem Zusammenhang wird auch der Begriff eines Blocks in Java erläutert.
Die folgenden Abschnitte erläutern die grundlegenden Anweisungen und Befehle zur Ablaufsteuerung, geordnet nach deren Wirkungsweise (Entscheidungsanweisungen, Schleifen und Sprungbefehle).
Achtung: Neben den hier vorgestellten Befehlen zur Ablaufsteuerung existiert eine weitere Gruppe solcher Befehle, die man im Zusammenhang mit Ausnahmen in Java verwendet. Diese Gruppe umfasst die Befehle **`try-catch-finally`** und **`throw`**. Wir gehen darauf in Kapitel 9 ein.

3.5.1 Anweisungen

Einige einfache Anweisungen lernten wir bereits kennen:

- Deklarationsanweisung, mit deren Hilfe wir Variablen vereinbaren;
- Zuweisungen, mit deren Hilfe wir Variablen Werte zuweisen;
- Methodenaufrufe, mit deren Hilfe wir zum Beispiel Ein- oder Ausgabeanweisungen realisierten.

Die beiden letztgenannten Anweisungen gehören zur Gruppe der Ausdrucksanweisungen. Der Name liegt darin begründet, dass bei einer Ausdrucksanweisung ein Ausdruck (ein Zuweisungsausdruck, eine Prä- oder Postfix-Operation mit ++ oder −− oder ein Methodenaufruf) durch Anhängen eines Semikolons zu einer Anweisung wird.
Daneben gibt es die sogenannte leere Anweisung, die einfach aus einem Semikolon besteht und tatsächlich auch an einigen Stellen (dort, wo syntaktisch eine Anweisung gefordert wird, wir aber keine Anweisung ausführen wollen) sinnvoll einsetzbar ist.

3.5.2 Blöcke und ihre Struktur

In der Programmiersprache Java bezeichnet ein **Block** eine Folge von Anweisungen, die durch { und } geklammert zusammengefasst sind. Solch ein Block kann immer dort, wo eine einzelne Anweisung erlaubt ist, verwendet werden, da ein Block im Prinzip *eine* Anweisung, nämlich eine zusammengesetzte Anweisung, darstellt. Dadurch ist es auch möglich, Blöcke zu schachteln.
Folgender Programmausschnitt enthält beispielsweise einen großen (äußeren) Block, in den zwei (innere) Blöcke geschachtelt sind.

```
{                  // Anfang des aeusseren Blocks
  int x = 5;               // Deklarationsanweisung und Zuweisung
  x++;                     // Postfix-Inkrement-Anweisung
  {                        // Anfang des ersten inneren Blocks
    long y;                      // Deklarationsanweisung
    y = x + 123456789;           // Zuweisung
    System.out.println(y);       // Ausgabeanweisung/Methodenaufruf
    ;                            // Leere Anweisung
  }                        // Ende des ersten inneren Blocks
  System.out.println(x);   // Ausgabeanweisung/Methodenaufruf
  {                        // Anfang des zweiten inneren Blocks
    double d;                    // Deklarationsanweisung
    d = x + 1.5;                 // Zuweisung
    System.out.println(d);       // Ausgabeanweisung/Methodenaufruf
  }                        // Ende des zweiten inneren Blocks
}                  // Ende des aeusseren Blocks
```

Anzumerken bleibt, dass Variablen, die wir in unserem Programm deklarieren, immer nur bis zum Ende des Blocks, in dem sie definiert wurden, gültig sind. Man spricht in diesem Zusammenhang auch vom **Gültigkeitsbereich** der Variablen. Beispielsweise können wir auf die Variable y im obigen Beispielprogramm, im äußeren Block und im zweiten inneren Block nicht mehr zugreifen, da diese mit der schließenden geschweiften Klammer nach der leeren Anweisung ihre Gültigkeit verloren hat. Man könnte auch sagen, die Variable y ist nur innerhalb des ersten inneren Blocks **gültig**.

3.5.3 Entscheidungsanweisung

3.5.3.1 Die `if`-Anweisung

Die wohl grundlegendste Entscheidungsanweisung vieler Programmiersprachen stellt die sogenannte **`if-else`**-Anweisung (deutsch: wenn-sonst) dar. Die Syntax dieser Anweisung sieht in Java allgemein wie folgt aus:

Syntaxregel

```
if («AUSDRUCK»)
  «ANWEISUNG»
else
  «ANWEISUNG»
```

Während der Programmausführung einer solchen **if-else**-Anweisung wird zunächst der Ausdruck ausgewertet, dessen Ergebnistyp **boolean** sein muss (es handelt sich also um einen logischen Ausdruck, z. B. einen Vergleich). Ist das Ergebnis **true**, so wird die unmittelbar nachfolgende Anweisung ausgeführt; ist es **false**, kommt die Anweisung nach **else** zur Ausführung. Danach wird mit der nächstfolgenden Anweisung fortgefahren. Es wird jedoch immer *genau eine* der beiden Anweisungen ausgeführt – die Anweisung nach **if** und die Anweisung nach **else** können also in einem Durchlauf der **if-else**-Anweisung niemals beide zur Ausführung kommen.
Will man keine Anweisungen durchführen, wenn der Ausdruck das Ergebnis **false** liefert, so kann man den **else**-Teil auch komplett weglassen.
Zu beachten ist, dass die beiden Anweisungen natürlich auch durch Blöcke ersetzt werden können, falls man die Ausführung mehrerer Anweisungen vom Ergebnis des logischen Ausdrucks abhängig machen will. Syntaktisch könnte das ganz allgemein also folgendermaßen aussehen.

Syntaxregel

```
if («AUSDRUCK») {
    «ANWEISUNG»
    ...
    «ANWEISUNG»
} else {
    «ANWEISUNG»
    ...
    «ANWEISUNG»
}
```

Auch hier gilt: Will man keine Anweisungen durchführen, wenn der Ausdruck das Ergebnis **false** liefert, so kann man den **else**-Teil auch komplett weglassen.
Achtung: Welche Anweisungen zu welchem Block gehören, wird ausschließlich durch die geschweiften Klammern festgelegt. Sind die Anweisungen im **if**- oder **else**-Teil nicht geklammert, gehört nur eine Anweisung in diesen Teil, egal, wie die Anweisungen eingerückt sind. Im Programmausschnitt

```
1 int x = IOTools.readInteger();
2 if (x == 0) {
3     System.out.println("x ist gleich 0");
4 } else
5     System.out.println("x ist ungleich 0, wir koennen dividieren");
6     System.out.println("1/x liefert " + 1/x);
7 System.out.println("Division durchgefuehrt");
```

gehören die Anweisungen in Zeile 6 und 7 nicht mehr zum **else**-Teil und werden deshalb auf jeden Fall ausgeführt, egal, welcher Wert für `x` eingelesen wird. Es empfiehlt sich daher, **if**- und **else**-Teile *immer* in Klammern zu setzen, auch wenn sie nur aus einer einzigen Anweisung bestehen. Nur so ist sofort ersichtlich, welche Anweisungen zu diesen Teilen gehören und welche nicht. Sie können

aber auch einen Editor verwenden, der sich selbstständig um die korrekte Einrückung kümmert. Werkzeuge mit dieser Funktion, dem sogenannten **Code Formatter** oder **Beautifier**, sind heutzutage in vielen Programmen direkt eingebaut und teilweise im Internet sogar als **Freeware** oder **Open Source** erhältlich.

3.5.3.2 Die `switch`-Anweisung

Eine weitere Entscheidungsanweisung stellt die **`switch`**–**`case`**–**`default`**-Kombination dar, mit deren Hilfe man in verschiedene Alternativen verzweigen kann. Die Syntax lautet allgemein wie folgt:

Syntaxregel

```
switch («AUSDRUCK») {
  case «KONSTANTE»:
    «ANWEISUNG»
    ...
    «ANWEISUNG»
    break;
  ...
  case «KONSTANTE»:
    «ANWEISUNG»
    ...
    «ANWEISUNG»
    break;
  default:
    «ANWEISUNG»
    ...
    «ANWEISUNG»
}
```

Die mit dem Wortsymbol **`case`** eingeleiteten Konstanten mit nachfolgendem Doppelpunkt legen dabei Einsprungmarken für den Programmablauf fest. Zwischen zwei solchen Einsprungmarken müssen nicht unbedingt Anweisungen stehen. Außerdem sind auch die Abbruchanweisungen (**`break`**;) sowie die Marke **`default`**: und die nachfolgenden Anweisungen optional.
Prinzipiell handelt es sich bei den Anweisungen im **`switch`**-Block um eine Folge von Anweisungen, von denen einige als Einsprungstellen markiert sind. Hier wird nämlich zunächst der Ausdruck ausgewertet, dessen Ergebnistyp ein **`byte`**, **`short`**, **`int`**, **`char`** oder `String` sein muss.[14] Daraufhin wird der Programmablauf bei genau der **`case`**-Marke, die als Konstante das Ergebnis des Ausdrucks enthält, fortgesetzt, bis auf eine **`break`**-Anweisung gestoßen wird, über die man die Ausführung der **`switch`**-Anweisung sofort beendet. Wird das Ergebnis des

[14] Daneben kann die **`switch`**-Anweisung auch für **`enum`**-Typen eingesetzt werden, mit denen wir uns aber erst in Abschnitt 10.1.3 beschäftigen werden.

Ausdrucks in keiner **case**-Anweisung gefunden, so wird die Programmausführung mit den Anweisungen nach der **default**-Marke fortgesetzt.
Zu beachten ist dabei, dass eben nicht nur die jeweils durch eine **case**-Marke markierten Anweisungen ausgeführt werden, sondern dass mit der Ausführung *aller* nachfolgenden Anweisungen fortgefahren wird und erst die nächste **break**-Anweisung die Ausführung der gesamten **switch**-Anweisung abbricht und den Programmablauf mit der ersten Anweisung außerhalb der **switch**-Anweisung fortsetzt. Dazu ein Beispiel:

```
int a, b;
switch (a) {
  case 1:
    b = 10;
  case 2:
  case 3:
    b = 20;
    break;
  case 4:
    b = 30;
    break;
  default:
    b = 40;
}
```

In dieser **switch**-Anweisung wird der Variablen b der Wert 20 zugewiesen, falls a den Wert 1, 2 oder 3 hat. Warum? Hat a den Wert 1, so wird zunächst an die erste **case**-Marke gesprungen und der Variablen b der Wert 10 zugewiesen. Danach fährt die Bearbeitung jedoch mit der nächsten Anweisung fort, da es nicht mit einer **break**-Anweisung explizit zum Verlassen der gesamten **switch**-Anweisung aufgefordert wurde. Nach der zweiten **case**-Marke wird gar kein Befehl ausgeführt, und die Bearbeitung setzt mit der Anweisung nach der dritten **case**-Marke fort. Jetzt wird der Variablen b der Wert 20 zugeordnet und anschließend die gesamte **switch**-Anweisung per **break**-Anweisung verlassen.
Enthält a zu Beginn der **switch**-Anweisung den Wert 4, so wird b der Wert 30 zugewiesen, in allen anderen Fällen enthält b nach Ausführung der **switch**-Anweisung schließlich den Wert 40.
Findet sich bei einer **switch**-Anweisung zur Laufzeit keine zum Ausdruck passende **case**-Marke und auch keine **default**-Marke, so bleibt die gesamte **switch**-Anweisung für den Programmablauf ohne Wirkung (wie etwa eine leere Anweisung, nur nimmt die Ausführung der **switch**-Anweisung mehr Zeit in Anspruch).
Nachfolgend noch ein Beispielprogramm, mit dem wir die Möglichkeit haben, für einen bestimmten Monat (als Text eingegeben) und für eine Jahresangabe (als ganzzahliger Wert eingegeben) die Anzahl der Tage im Monat und die laufende Nummer des Monats im Jahr bestimmen zu lassen. Ein Programmablauf könnte etwa so aussehen:

Konsole

```
Monat (als Zeichenkette): Februar
Jahr (positive ganze Zahl): 2020
```

```
Der Monat Februar des Jahres 2020 hat 29 Tage.
Der Monat Februar ist der 2. Monat des Jahres.
```

Realisiert haben wir dies in unserem Programm mit den Methoden `monatsZahl` und `tageImMonat`. Beide Methoden verwenden im Kopf der **`switch`**-Anweisung den Ausdruck `monat.toLowerCase()`, mit dem wir dafür sorgen, dass wir uns in den **`case`**-Labels nur um die klein geschriebene Form des Monatsnamens kümmern müssen. Mit dieser Art von `String`-Methodenaufrufen werden wir uns später in Abschnitt 5.5.2 noch genauer beschäftigen.

```
1  import Prog1Tools.IOTools;
2  public class StringSwitchDemo {
3    public static int monatsZahl(String monat) {
4        int zahl = 0;
5        if (monat == null) {
6            return zahl;
7        }
8        switch (monat.toLowerCase()) {
9            case "januar":
10             zahl =  1;
11             break;
12           case "februar":
13             zahl =  2;
14             break;
15           case "maerz":
16             zahl =  3;
17             break;
18           case "april":
19             zahl =  4;
20             break;
21           case "mai":
22             zahl =  5;
23             break;
24           case "juni":
25             zahl =  6;
26             break;
27           case "juli":
28             zahl =  7;
29             break;
30           case "august":
31             zahl =  8;
32             break;
33           case "september":
34             zahl =  9;
35             break;
36           case "oktober":
37             zahl = 10;
38             break;
39           case "november":
40             zahl = 11;
41             break;
42           case "dezember":
43             zahl = 12;
44             break;
```

```
45        }
46        return zahl;
47    }
48
49    public static int tageImMonat(String monat, int jahr) {
50      int tage = 0;
51      if (monat == null) {
52          return tage;
53      }
54      switch (monat.toLowerCase()) {
55          case "februar":
56            if ((jahr%4 != 0) || ((jahr%100 == 0) && (jahr%400 != 0))) {
57                tage =  28;
58            } else {
59                tage = 29;
60            }
61            break;
62          case "april":
63          case "juni":
64          case "september":
65          case "november":
66            tage =  30;
67            break;
68          case "januar":
69          case "maerz":
70          case "mai":
71          case "juli":
72          case "august":
73          case "oktober":
74          case "dezember":
75            tage =  31;
76            break;
77      }
78      return tage;
79    }
80    public static void main(String[] args) {
81      String m = IOTools.readString("Monat (als Zeichenkette): ");
82      int j = IOTools.readInt("Jahr (positive ganze Zahl): ");
83
84      int t = tageImMonat(m,j);
85      if (t == 0) {
86          System.out.println("Unzulaessiger Monat");
87      } else if (j <= 0) {
88          System.out.println("Unzulaessiges Jahr");
89      } else {
90          System.out.println("Der Monat " + m + " des Jahres " + j +
91                             " hat " + t + " Tage.");
92      }
93
94      int z = monatsZahl(m);
95      if (z == 0) {
96          System.out.println("Unzulaessiger Monat");
97      } else {
98          System.out.println("Der Monat " + m + " ist der " + z +
99                             ". Monat des Jahres.");
```

```
100         }
101       }
102   }
```

3.5.4 Wiederholungsanweisungen, Schleifen

Eine weitere Gruppe der Befehle zur Ablaufsteuerung stellen die sogenannten **Wiederholungsanweisungen** bzw. **Schleifen** dar. Wie die Namen dieser Anweisungen bereits deutlich machen, können damit eine Anweisung bzw. ein Block von Anweisungen mehrmals hintereinander ausgeführt werden.

3.5.4.1 Die `for`-Anweisung

Der erste Vertreter dieser Schleifen ist die **`for`**-Anweisung. Ihre Syntax lautet:

Syntaxregel

```
for («INITIALISIERUNG» ; «AUSDRUCK» ; «UPDATELISTE»)
  «ANWEISUNG»
```

bzw.

Syntaxregel

```
for («INITIALISIERUNG» ; «AUSDRUCK» ; «UPDATELISTE») {
  «ANWEISUNG»
  ...
  «ANWEISUNG»
}
```

Dabei werden zunächst im Teil «INITIALISIERUNG» eine oder mehrere (typgleiche) Variablen vereinbart und initialisiert und daraufhin der Ausdruck, dessen Ergebnistyp wiederum vom Typ **`boolean`** sein muss, ausgewertet. Ist sein Wert **`true`**, so werden die Anweisung bzw. der Anweisungsblock (auch **Rumpf** genannt) ausgeführt und danach zusätzlich die Anweisungen in der Update-Liste (eine Kommaliste von Anweisungen) ausgeführt. Dies wird so lange wiederholt, bis der Ausdruck den Wert **`false`** liefert.
Dazu ein Beispiel:

```
for (int i = 0; i < 10; i++)
  System.out.println(i);
```

Dieses Programmstück macht nichts anderes, als die Zahlen `0` bis `9` zeilenweise auf dem Bildschirm auszudrucken. Wie funktioniert das? Zunächst wird die Initialisierungsanweisung `int i = 0;` ausgeführt, d. h. die Variable `i` wird deklariert und mit dem Wert `0` initialisiert. Als Nächstes wird der Ausdruck `i < 10` ausgewertet – dies ergibt **`true`**, da `i` ja gerade den Wert `0` hat, die Anweisung `System.out.println(i);` wird also ausgeführt und druckt die Zahl `0` auf den Bildschirm. Nun wird zunächst die Update-Anweisung `i++` durchgeführt,

die den Wert von `i` um eins erhöht, und danach wieder der Ausdruck `i < 10` ausgewertet, was auch jetzt wieder **`true`** als Ergebnis liefert – die Anweisung `System.out.println(i);` kommt somit erneut zur Ausführung. Dieses Spiel setzt sich so lange fort, bis der Ausdruck `i < 10` das Ergebnis **`false`** liefert, was genau dann zum ersten Mal der Fall ist, wenn die Variable `i` den Wert `10` angenommen hat, worauf die Anweisung `System.out.println(i);` nicht mehr ausgeführt und die Schleife beendet wird.
Analog zu diesem Beispiel lässt sich auch die nachfolgende Schleife programmieren.

```
for (int i = 9; i >= 0; i--)
  System.out.println(i);
```

Hier werden nun, man ahnt es schon, wieder die Zahlen `0` bis `9` auf dem Bildschirm ausgegeben, diesmal jedoch in umgekehrter Reihenfolge.
Anzumerken bleibt, dass es – wie schon bei **`if-else`**-Anweisungen – auch hier sinnvoll ist, die zur Schleife gehörigen Anweisungen *immer* als Block zu klammern, auch wenn nur eine Anweisung existiert, um möglichen Verwechslungen vorzubeugen.
Achtung: Die drei syntaktischen Bestandteile «INITIALISIERUNG», «AUSDRUCK» und «UPDATELISTE» der **`for`**-Anweisung können auch (einzeln oder zusammen) entfallen. Solche spezielle **`for`**-Schleifen verzichten auf eine Initialisierung zu Beginn bzw. ein Update nach jedem Schleifendurchlauf. Fehlt der logische Ausdruck, setzt der Compiler ein **`true`** dafür ein, sodass diese Schleife gar nicht abbrechen würde!

3.5.4.2 Vereinfachte `for`-Schleifen-Notation

Es gibt in Java auch eine vereinfachte Notation für **`for`**-Schleifen, die sich allerdings erst in Verbindung mit strukturierten Datentypen, wie wir sie zum Beispiel in Kapitel 4 kennenlernen werden, sinnvoll einsetzen lässt. Der Kopf der **`for`**-Schleife kann dabei gemäß der Syntax

Syntaxregel

```
for («TYP» «VARIABLENNAME» : «AUSDRUCK»)
```

formuliert werden. Ohne an dieser Stelle genauer darauf einzugehen, von welchem Datentyp «AUSDRUCK» sein muss, sei zumindest erwähnt, dass wir beispielsweise einen **`for`**-Schleifen-Kopf der Form

```
for (int x : w)
```

als *„für jedes* `x` *in* `w`“ lesen können. Das heißt: die Variable `x` nimmt nacheinander alle in `w` vorkommenden Werte in der durch `w` bestimmten Reihenfolge an. Auf weitere Details gehen wir in den Abschnitten 4.1.9 und 11.7.2 ein.

3.5.4.3 Die `while`-Anweisung

Einen weiteren Schleifentyp stellt die „abweisende“ **`while`-Schleife** dar. Als „abweisend“ wird sie deshalb bezeichnet, weil hier, bevor irgendwelche Anweisungen zur Ausführung kommen, zunächst ein logischer Ausdruck geprüft wird. Die Syntax lautet:

Syntaxregel

```
while («AUSDRUCK»)
  «ANWEISUNG»
```

bzw.

Syntaxregel

```
while («AUSDRUCK») {
  «ANWEISUNG»
  ...
  «ANWEISUNG»
}
```

Hier wird also zunächst der Ausdruck ausgewertet (Ergebnistyp **`boolean`**) und – solange dieser den Wert **`true`** liefert – die Anweisung bzw. der Anweisungsblock ausgeführt und der Ausdruck erneut berechnet.
Dazu obiges Beispiel für die **`for`**-Anweisung, jetzt mit der **`while`**-Anweisung:

```
int i = 0;
while (i < 10) {
  System.out.println(i);
  i++;
}
```

Anzumerken bleibt auch hier wieder, dass es sinnvoll ist, die zur Schleife gehörigen Anweisungen *immer* als Block zu klammern, auch wenn nur eine Anweisung existiert, um möglichen Verwechslungen vorzubeugen.

3.5.4.4 Die `do`-Anweisung

Den dritten und letzten Schleifentyp in Java stellt die „nicht-abweisende“ **`do`-Schleife** dar. Als „nicht abweisend“ wird diese wiederum deshalb bezeichnet, weil hier die Anweisungen auf jeden Fall zur Ausführung kommen, bevor ein logischer Ausdruck geprüft wird. Die Syntax lautet:

Syntaxregel

```
do
  «ANWEISUNG»
while («AUSDRUCK»);
```

bzw.

Syntaxregel

```
do {
  «ANWEISUNG»
  ...
  «ANWEISUNG»
} while («AUSDRUCK»);
```

Hier werden also die Anweisung bzw. der Anweisungsblock zunächst einmal ausgeführt und danach der Ausdruck ausgewertet. Solange dieser den Wert **true** liefert, wird das Ganze wiederholt. Der Unterschied zur **while**-Schleife ist somit die Tatsache, dass bei der abweisenden Schleife der logische Ausdruck *noch vor der ersten Ausführung* einer Anweisung aus dem Schleifenrumpf überprüft wird, während bei der nicht-abweisenden **do**-Schleife der Ausdruck *erst nach der ersten Durchführung* der Anweisung(en) ausgewertet wird. Es kann daher vorkommen, dass bei der abweisenden Schleife gar keine Anweisung des Schleifenrumpfs ausgeführt wird, während bei der nicht-abweisenden Schleife auf jeden Fall mindestens einmal etwas ausgeführt wird.
Dazu obiges Beispiel für die **while**-Anweisung jetzt mit der **do**-Anweisung:

```
int i = 0;
do {
  System.out.println(i);
  i++;
} while (i < 10);
```

Hier nochmals der Hinweis, dass es sinnvoll ist, die zur Schleife gehörigen Anweisungen *immer* als Block zu klammern, auch wenn nur eine Anweisung existiert, um möglichen Verwechslungen vorzubeugen.

3.5.4.5 Endlosschleifen

Beim Programmieren von Schleifen ist es (gewollt oder unbeabsichtigt) möglich, sogenannte **Endlosschleifen** (auch **unendliche Schleifen** genannt) zu formulieren. Die Namensgebung ist durch die Tatsache begründet, dass die Anweisungen des Schleifenrumpfs unendlich oft zur Ausführung kommen. Beispiele für bewusst formulierte Endlosschleifen wären etwa

```
for (int i=1; ; i++) {
  System.out.println(i);
}
```

oder

```
while (true) {
  System.out.println("Nochmal!");
}
```

Um ungewollte Endlosschleifen zu vermeiden, ist eine gewisse Vorsicht bei der Formulierung der logischen Ausdrücke, die für den Abbruch der Schleife sorgen, geboten. Außerdem müssen die Anweisungen innerhalb des Schleifenrumpfs die

Operanden des logischen Ausdrucks nach endlich vielen Schritten derart verändern, dass der Ausdruck den Wert **false** liefert und die Schleife dadurch zum Ende kommt. Bei den beiden nachfolgenden Beispielen haben sich leider Programmierfehler eingeschlichen, sodass obige Forderung nicht erfüllt ist. In der **do**-Schleife

```
int i=0;
do {
    System.out.println("Nochmal!");
} while (i < 10);
```

wurde vergessen, die Variable i bei jedem Durchlauf zu erhöhen. In der **for**-Schleife

```
for (int i=0; i<10; i++) {
    System.out.println("Nochmal!");
    i--;
}
```

neutralisiert leider die Dekrementierung von i in der letzten Anweisung des Schleifenrumpfs die Inkrementierung von i in der Update-Liste.

3.5.5 Sprungbefehle und markierte Anweisungen

Zuletzt wollen wir uns noch mit der Klasse der sogenannten **Sprungbefehle** vertraut machen, mit denen man z. B. aus Schleifen herausspringen und diese damit vorzeitig beenden kann. Für diejenigen, die bereits Erfahrung in Programmiersprachen wie Basic oder C++ gesammelt haben, eine kleine Warnung vorweg: In der Sprache Java gibt es im Gegensatz zu anderen Programmiersprachen *keine* **goto**-Anweisung – und das ist auch gut so! Überhaupt sollte man die hier vorgestellten Befehle, insbesondere **break** und **continue**, nur mit Bedacht einsetzen, denn nichts ist unübersichtlicher (und damit fehleranfälliger) als Programme, in denen ständig wild hin- und hergesprungen wird.
Die Anweisung **break** haben wir schon in Zusammenhang mit der **switch**-Anweisung kennengelernt. Sie dient ganz allgemein dazu, den gerade in Ausführung befindlichen *innersten* Block bzw. die gerade in Ausführung befindliche *innerste* Schleife zu unterbrechen und mit der Anweisung, die direkt nach dem Block bzw. der Schleife folgt, fortzufahren.
In Java ist es aber auch bei geschachtelten Blöcken und Schleifen möglich, diese gezielt vorzeitig abzubrechen. Dazu kann man eine sogenannte **Marke** (bestehend aus einem Bezeichner, gefolgt von einem Doppelpunkt) verwenden und einen Block bzw. eine Schleife markieren. Kennzeichnet man zum Beispiel eine **while**-Schleife in der Form

```
marke:
  while (n>3) {
    ...
  }
```

so kann man in deren Anweisungsteil weitere Schleifen und Blöcke schachteln und aus diesen inneren Schleifen oder Blöcken mit dem Befehl

```
break marke;
```

herausspringen und die komplette **while**-Schleife abbrechen.
Wir wollen uns dazu folgendes Beispiel ansehen:

```
1  dieda:
2    for (int k = 0; k < 5; k++) {
3      for (int i = 0; i < 5; i++) {
4        System.out.println("i-Schleife i = " + i);
5        if (k == 3)
6          break dieda;
7        else
8          break;
9      }
10     System.out.println("k-Schleife k = " + k);
11   }
12 System.out.println("jetzt ist Schluss");
```

Aufgrund der **break**-Anweisungen in der innersten **for**-Schleife wird `i` nie größer als 0. Da man für `k = 3` die **break**-Anweisung mit Marke `dieda` verwendet, wird dabei sogar die `k`-Schleife beendet. Auf der Konsole gibt dieses Programmstück somit Folgendes aus:

Konsole

```
i-Schleife i = 0
k-Schleife k = 0
i-Schleife i = 0
k-Schleife k = 1
i-Schleife i = 0
k-Schleife k = 2
i-Schleife i = 0
jetzt ist Schluss
```

Die Anweisung **continue** entspricht der **break**-Anweisung in dem Sinne, dass der aktuelle Schleifendurchlauf sofort beendet ist. Allerdings ist nicht die gesamte Schleifenanweisung beendet, sondern es wird mit dem nächsten Schleifendurchlauf weitergemacht. Bei **for**-Schleifen wird also durch **continue** zu den Anweisungen in der Update-Liste verzweigt.
Auch hierzu wollen wir uns ein Beispiel ansehen:

```
1  for (int i=-10; i<=10; i++){
2    if (i == 0)
3      continue;
4    System.out.println ("Division von 1 durch " + i +
5                        " ergibt " + 1/i);
6  }
```

Hier wird durch die Verwendung von **continue** vermieden, dass eine ganzzahlige Division durch Null durchgeführt wird.
Wie **break** kann auch **continue** zusammen mit einer Marke verwendet werden, sodass z. B. mit dem nächsten Schleifendurchlauf einer umgebenden (natürlich entsprechend markierten) Schleife fortgefahren werden kann.

Eine Sonderstellung unter den Sprungbefehlen nimmt die **return**-Anweisung ein. Sie dient dazu, aufgerufene Methoden zu beenden und eventuell einen Rückgabewert an die aufrufende Umgebung weiterzugeben. Da wir jedoch noch nicht wissen, was Methoden sind und wie sie aufgerufen werden, werden wir uns später noch einmal ausführlich mit der **return**-Anweisung befassen.

3.5.6 Zusammenfassung

Wir haben gesehen, wie wir Anweisungen zur Ablaufsteuerung dazu einsetzen können, um zu bestimmen, ob und wann andere Anweisungen in unserem Programm ausgeführt werden. Wir haben Blöcke, Entscheidungsanweisungen, Schleifen und Sprungbefehle kennengelernt.

3.5.7 Übungsaufgaben

Aufgabe 3.16

Gegeben sei folgender Ausschnitt aus einem Programm:

```
int i = 20;
while (i > 0) {
  System.out.println(i);
  i -= 2;
}
```

Was bewirkt die Schleife? Wie lautet eine **for**-Schleife mit gleicher Ausgabe?

Aufgabe 3.17

Was bewirken die Zeilen:

```
while (true) {
  System.out.println("Aloha");
}
```

Aufgabe 3.18

Bestimmen Sie die Ausgabe des nachfolgenden Java-Programms:

```
1  public class BreakAndContinue {
2    public static void main(String args[]) {
3      for(int i = 0; i < 100; i++) {
4        if(i == 74) break;
5        if(i % 9 != 0) continue;
6        System.out.println(i);
7      }
8      int i = 0;
9      while(true) {      // Endlosschleife ?
10       i++;
11       int j = i * 30;
12       if(j == 1260) break;
13       if(i % 10 != 0) continue;
```

```
14          System.out.println(i);
15        }
16      }
17    }
```

Aufgabe 3.19

Der Algorithmus

1. Lies den Wert von n ein.
2. Setze i auf 3.
3. Solange $i < 2n$, wiederhole:
   a. Erhöhe i um 1.
   b. Gib $\frac{1}{2i+1}$ aus.

soll auf drei verschiedene Arten implementiert werden: Schreiben Sie jeweils ein Java-Programmstück, das diesen Algorithmus als `while`-, als `for`- und als `do-while`-Schleife realisiert. Sämtliche Programmstücke sollen die gleichen Ausgaben erzeugen!

Aufgabe 3.20

Sie wollen ein Schachbrett nummerieren in der Form

Konsole

```
1  2  3  4  5  6  7  8
2  3  4  5  6  7  8  9
3  4  5  6  7  8  9 10
4  5  6  7  8  9 10 11
5  6  7  8  9 10 11 12
6  7  8  9 10 11 12 13
7  8  9 10 11 12 13 14
8  9 10 11 12 13 14 15
```

Formulieren Sie eine geschachtelte **`for`**-Schleife, die eine entsprechend formatierte Ausgabe erzeugt.

Aufgabe 3.21

An nachfolgendem Beispiel sehen Sie schlechten Programmierstil bei Schleifen.

```
int i, j;
for (i=1; i<=10; i++) { // Schleife A
  System.out.println("A1:  i = " + i);
  i = 5;
  System.out.println("A2:  i = " + i);
  for (i=7; i<=20; i++) { // Schleife B
    System.out.println("B1:  i = " + i);
```

```
            i = i + 2;
            System.out.println("B2:  i = " + i);
          }
        }
```

Könnten Sie auf Anhieb sagen, wie oft welche Schleife durchlaufen wird? Was wird ausgegeben?

Aufgabe 3.22

Das nachfolgende Java-Programm ist syntaktisch korrekt und könnte somit übersetzt und ausgeführt werden. Es enthält jedoch vier Beispiele für logische Fehler, die von einem schlechten Programmierer eingeschleppt wurden. Da diese beim Programmablauf teilweise zu einem Abbruch führen würden, sollten Sie die Fehler finden und korrigieren. Versuchen Sie, zu diesem Zwecke keinen Compiler zu benutzen, und finden Sie die Fehler, ohne das Programm auch nur ein einziges Mal auszuführen.

```
1   public class Falsch {
2     public static void main (String[] args) {
3       int x = 0, y = 4;
4                                                   // Beispiel A
5       if (x < 5)
6           if (x < 0)
7               System.out.println("x < 0");
8       else
9           System.out.println("x >= 5");
10                                                  // Beispiel B
11      if (x > 0)
12          System.out.println("ok! x > 0");
13          System.out.println("1/x = " + (1/x));
14                                                  // Beispiel C
15      if (x > 0);
16          System.out.println("1/x = " + (1/x));
17                                                  // Beispiel D
18      if (y > x) {
19           // vertausche x und y
20           x = y;
21           y = x;
22      }
23      System.out.println("x = " + x + "     y = " + y);
24    }
25  }
```

Aufgabe 3.23

Gegeben sei das nachfolgende Java-Programm.

```
1  import Prog1Tools.IOTools;
2  public class Irgendwas {
3    public static void main(String [] args) {
4      double a, b, c, d, e;
5      a = IOTools.readDouble("a = ");
```

```
6       b = IOTools.readDouble("b = ");
7       c = IOTools.readDouble("c = ");
8       d = IOTools.readDouble("d = ");
9       if (b > a)
10        if (c > b)
11          if (d > c)
12            e = d;
13          else
14            e = c;
15        else
16          if (d > b)
17            e = d;
18          else
19            e = b;
20      else
21        if (c > a)
22          if (d > c)
23            e = d;
24          else
25            e = c;
26        else
27          if (d > a)
28            e = d;
29          else
30            e = a;
31      System.out.println("e = " + e);
32    }
33  }
```

Welcher Wert e wird von diesem Programm berechnet und ausgegeben?
Überlegen Sie sich ein deutlich kürzeres Programmstück, das mit nur drei **if**-Anweisungen auskommt, aber das Gleiche leistet.

Aufgabe 3.24

Schreiben Sie ein Programm, das mit Hilfe geschachtelter Schleifen ein aus $\star$-Zeichen zusammengesetztes Dreieck auf der Konsole ausgibt. Die Benutzerin bzw. der Benutzer soll vorher nach der Anzahl der Zeilen gefragt werden.
Beispiel für den Programmablauf:

Konsole

```
Anzahl der Zeilen: 4
*
**
***
****
```

Aufgabe 3.25

Schreiben Sie ein Java-Programm, das eine **int**-Zahl z mit $0 < z < 10000$ einliest, ihre Quersumme berechnet und die durchgeführte Berechnung sowie den Wert der Quersumme wie nachfolgend dargestellt ausgibt.

Konsole

```
Positive ganze Zahl eingeben: 2345
Die Quersumme ergibt sich zu: 5 + 4 + 3 + 2 = 14
```

Aufgabe 3.26

Ein neuer Science-Fiction-TV-Sender will sein Programmschema nur noch in galaktischer Zeitrechnung angeben. Dazu sollen Sie ein Java-Programm schreiben, das eine Datums- und Uhrzeitangabe in Erdstandardzeit in eine galaktische Sternzeit umrechnet.
Eine Sternzeit wird als Gleitkommazahl angegeben, wobei die Vorkommastellen die Tageszahl (das Datum) und die Nachkommastellen die galaktischen Milli-Einheiten (die Uhrzeit) angeben. Der Tag 1 in der galaktischen Zeitrechnung entspricht gerade dem 1.1.1111 auf der Erde. Ein Galaxis-Tag hat 1000 Milli-Einheiten und dauert 1440 Erdminuten, also zufälligerweise genau 24 Stunden. Die Sternzeit 5347.789 entspricht somit gerade dem 25.8.1125, 18.57 Uhr Erdstandardzeit.
In Ihrem Programm müssen Sie zu einer durch die Werte `jahr`, `monat`, `tag`, `stunde` und `minute` vorgegebenen Erd-Datum- und Erd-Zeit-Angabe zunächst die Anzahl der Tage bestimmen, die seit dem 1.1.1111 bereits vergangen sind. Dabei brauchen Schaltjahre nicht berücksichtigt zu werden. Zu dieser Zahl muss dann der gebrochene Zeit-Anteil addiert werden, der sich ergibt, wenn man die durch die Uhrzeit festgelegten Erdminuten in Bruchteile eines Tages umrechnet und diese auf drei Ziffern nach dem Dezimalpunkt rundet.
Testen Sie Ihr Programm auch an folgendem Beispiel:

Konsole

```
Erdzeit 11.11.2111, 11.11 Uhr
   entspricht der Sternzeit 365315.465.
```

Aufgabe 3.27

Schreiben Sie ein Programm, das eine positive ganze Zahl einliest, sie in ihre Ziffern zerlegt und die Ziffern in umgekehrter Reihenfolge als Text ausgibt. Verwenden Sie dabei eine **`while`**-Schleife und eine **`switch`**-Anweisung.
Beispiel für den Programmablauf:

Konsole

```
Positive ganze Zahl: 35725
Zerlegt rueckwaerts: fuenf zwei sieben fuenf drei
```

Aufgabe 3.28

Schreiben Sie ein Programm, das unter Verwendung einer geeigneten Schleife eine ganze Zahl von der Tastatur einliest und deren Vielfache (für die Faktoren 1 bis 10) ausgibt. Programmablauf-Beispiel:

Konsole

```
Geben Sie eine Zahl ein: 3
Die Vielfachen: 3 6 9 12 15 18 21 24 27 30
```

Aufgabe 3.29

Schreiben Sie ein Programm zur Zinseszinsberechnung. Nach Eingabe des anzulegenden Betrages, des Zinssatzes und der Laufzeit der Geldanlage soll der Wert der Investition nach jedem Jahr ausgegeben werden. Programmablauf-Beispiel:

Konsole

```
Anzulegender Geldbetrag in Euro: 100
Jahreszins (z. B. 0.1 fuer 10 Prozent): 0.06
Laufzeit (in Jahren): 4
Wert nach 1 Jahren: 106.0
Wert nach 2 Jahren: 112.36
Wert nach 3 Jahren: 119.1016
Wert nach 4 Jahren: 126.247696
```

Aufgabe 3.30

Programmieren Sie ein Zahlenraten-Spiel. Im ersten Schritt soll die Benutzerin bzw. der Benutzer begrüßt und kurz über die Regeln des Spiels informiert werden. Danach soll durch die Anweisung

```
int geheimZahl = (int) (99 * Math.random() + 1);
```

eine Zufallszahl `geheimZahl` zwischen `0` und `100` generiert werden.[15] Die Benutzerin bzw. der Benutzer des Programms soll nun versuchen, diese Zahl zu erraten. Programmieren Sie dazu eine Schleife, in der in jedem Durchlauf jeweils

- darüber informiert wird, um den wievielten Rateversuch es sich handelt,
- ein Rateversuch eingegeben werden kann und
- darüber informiert wird, ob die geratene Zahl zu groß, zu klein oder korrekt geraten ist.

Diese Schleife soll so lange durchlaufen werden, bis die Zahl erraten ist.
Beispiel für den Programmablauf:

Konsole

```
Willkommen beim Zahlenraten.
Ich denke mir eine Zahl zwischen 1 und 100. Rate diese Zahl!
1. Versuch: 50
Meine Zahl ist kleiner!
2. Versuch: 25
```

[15] Die Methode `Math.random` liefert eine Zufallszahl x vom Typ **`double`**, für die gilt: $0 \leq x < 1$. Die Klasse `Math` und ihre Methoden werden in Abschnitt 5.4.2 behandelt.

```
Meine Zahl ist kleiner!
3. Versuch: 12
Du hast meine Zahl beim 3. Versuch erraten!
```

Aufgabe 3.31

Schreiben Sie ein Java-Programm, das eine einzulesende ganze Dezimalzahl d in eine Binärzahl b umrechnet und ausgibt. Dabei soll d mit Hilfe des Datentyps **short** und b mit Hilfe des Datentyps **long** dargestellt werden, wobei b nur die Ziffern 0 und 1 enthalten darf. Die **long**-Zahl 10101 soll also z. B. der Binärzahl $10101_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 21_{10}$ entsprechen.
Verwenden Sie (bei geeigneter Behandlung des Falles $d < 0$) den Algorithmus:

1. Setze $b = 0$ und $m = 1$.
2. Solange $d > 0$ gilt, führe folgende Schritte durch:

   Addiere $(d \,\%\, 2) \cdot m$ zu b.

   Setze $d = d/2$ und multipliziere m mit 10.
3. b enthält nun die gesuchte Binärzahl.

Beispiel für den Programmablauf:

Konsole

```
   Dezimalzahl: 21
als Binaerzahl: 10101
```

Wie müsste man den Algorithmus ändern, wenn man z. B. ins Oktalsystem umrechnen wollte?

Aufgabe 3.32

Schreiben Sie ein Programm, das einen Weihnachtsbaum mit Hilfe von **for**-Schleifen zeichnet. Lesen Sie die gewünschte Höhe des Baumes von der Tastatur ein, und geben Sie entsprechend einen Baum wie im folgenden Beispiel aus:

Konsole

```
Anzahl der Zeilen: 5
        *
       ***
      *****
     *******
    *********
        I
```

Aufgabe 3.33

Zwei verschiedene natürliche Zahlen a und b heißen *befreundet*, wenn die Summe der (von a verschiedenen) Teiler von a gleich b ist und die Summe der (von b verschiedenen) Teiler von b gleich a ist.
Ein Beispiel für ein solches befreundetes Zahlenpaar ist $(a, b) = (220, 284)$, denn $a = 220$ hat die Teiler $1, 2, 4, 5, 10, 11, 20, 22, 44, 55, 110$ (und $220 = a$), und es gilt

$$1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284 = b.$$

Weiterhin hat $b = 284$ die Teiler $1, 2, 4, 71, 142$ (und $284 = b$), und es gilt

$$1 + 2 + 4 + 71 + 142 = 220 = a.$$

Schreiben Sie ein Java-Programm, das jeweils zwei Zahlen einliest und entscheidet, ob diese miteinander befreundet sind. Arbeiten Sie mit einer geeigneten Schleife, in der alle Teiler einer Zahl bestimmt und aufsummiert werden.
Der Programmablauf könnte in etwa wie folgt aussehen:

Konsole

```
Erste Zahl  : 220
Zweite Zahl : 284
Die beiden Zahlen sind miteinander befreundet!
Erste Zahl  : 10744
Zweite Zahl : 10856
Die beiden Zahlen sind miteinander befreundet!
```

Aufgabe 3.34

Schreiben Sie ein Java-Programm, das zu einem beliebigen Datum den zugehörigen Wochentag ausgibt. Ein Datum soll jeweils durch drei ganzzahlige Werte t (Tag), m (Monat) und j (Jahr) vorgegeben sein. Schreiben Sie Ihr Programm unter Berücksichtigung der folgenden Teilschritte:

a) Vereinbaren Sie drei Variablen `t`, `m` und `j` vom Typ **`int`**, und lesen Sie für diese Werte ein.

b) Berechnen Sie den Wochentag h nach folgendem Algorithmus (% bezeichnet dabei den Java-Rest-Operator):

   1. Falls $m \leq 2$ ist, erhöhe m um 10 und erniedrige j um 1, andernfalls erniedrige m um 2.
   2. Berechne die ganzzahligen Werte $c = j/100$ und $y = j \ \% \ 100$.
   3. Berechne den ganzzahligen Wert

$$h = (((26 \cdot m - 2)/10) + t + y + y/4 + c/4 - 2 \cdot c) \ \% \ 7.$$

4. Falls $h < 0$ sein sollte, erhöhe h um 7.

Anschließend hat h einen Wert zwischen 0 und 6, wobei die Werte 0, 1, ..., 6 den Tagen Sonntag, Montag, ..., Samstag entsprechen.

c) Geben Sie das Ergebnis in der folgenden Form aus:

```
Der 24.12.2019 ist ein Dienstag.
```

Aufgabe 3.35

Der nachfolgende Algorithmus geht auf Carl Friedrich Gauß (siehe [51]) zurück und berechnet das Datum des Ostersonntags im Jahr j (gültig vom Jahr 1 bis zum Jahr 8202). Es bezeichnen / und % die üblichen ganzzahligen Divisionsoperatoren von Java.

1. Berechne $a = j \,\%\, 19$, $b = j \,\%\, 4$ und $c = j \,\%\, 7$.
2. Bestimme $m = (8 \cdot (j/100) + 13)/25 - 2$, $s = j/100 - j/400 - 2$,
   $m = (15 + s - m) \,\%\, 30$, $n = (6 + s) \,\%\, 7$.
3. Bestimme d und e wie folgt:
   (a) Setze $d = (m + 19 \cdot a) \,\%\, 30$.
   (b) Falls $d = 29$ ist, setze $d = 28$,
   andernfalls: falls $d = 28$ und $a \geq 11$ ist, setze $d = 27$.
   (c) Setze $e = (2 \cdot b + 4 \cdot c + 6 \cdot d + n) \,\%\, 7$.
4. Nun können der tag und der $monat$ bestimmt werden:
   (a) $tag = 21 + d + e + 1$
   (b) Falls $tag > 31$ ist, setze $tag = tag \,\%\, 31$ und $monat = 4$,
   andernfalls setze $monat = 3$.

Schreiben Sie ein Java-Programm, das den obigen Algorithmus durchführt und das Datum des Ostersonntags für ein einzulesendes Jahr berechnet und ausgibt. Beispiel für eine Ausgabezeile:

Konsole

```
Im Jahr 2019 ist der Ostersonntag am 21.4.
```