
PEARSON
Education
jetzt lerne ich
Start ohne Vorwissen
Java 6
Komplettes Starterkit für den erfolgreichen Einstieg in die Programmierung
DIRK LOUIS PETER MÜLLER
Markt+Technik
CD
inklusive aller Beispiele und Tools (JDK und Eclipse)

Kapitel 3

jetzt lerne ich

Von Daten, Operatoren und Objekten

Anscheinend haben Sie dieses Buch nach dem vorangehenden Kapitel doch nicht in den Altpapiercontainer geworfen, und das ist auch gut so. Denn der schwierigste, weil verwirrendste Teil liegt bereits hinter uns! Der erste Kontakt mit einer Programmiersprache ist nicht gerade leicht – und wenn dies schon für Sprachen wie Pascal und C gilt, so ist es erst recht wahr für Java mit seinem konsequent objektorientierten Aufbau. Viele unbekannte Konzepte, seltsame Schreibweisen und Begriffe prasseln da auf den Anfänger ein und die nagende Frage taucht auf: soll ich mir das antun?

»Der Zweifel ist´s, der Gutes böse macht!« (Goethe, Iphigenie auf Tauris)

Halten Sie also noch ein bisschen durch, ab diesem Kapitel wird alles leichter. Fortan werden wir systematisch an die Sache herangehen, wir werden die aufregende Welt der Java-Programmierung Stück um Stück für uns erobern und wir werden unserer eigenen Kreativität als Programmierer Tür und Tor öffnen.

Sie lernen in diesem Kapitel,

- was Variablen sind,
- warum jede Variable einem Datentyp angehört,
- wie man Variablen benutzt,
- wie man Operatoren auf Variablen anwendet,
- was Variablen von Konstanten unterscheidet,
- wie Datentypen umgewandelt werden,
- was objektorientierte Programmierung ausmacht,
- was Klassen sind,
- wie man mit Klassen programmiert,
- was Arrays sind,
- wie man die von Java vordefinierten Klassen verwendet.

3.1 Variablen und Anweisungen

Die Aufgabe eines jeden Computerprogramms ist die Verarbeitung von irgendwelchen Informationen, die im Computerjargon meist Daten genannt werden. Das können Zahlen sein, aber auch Buchstaben, ganze Texte oder Bilder und Zeichnungen. Dem Rechner ist diese Unterscheidung gleich, da er letztlich alle Daten in Form von endlosen Zahlenkolonnen in Binärdarstellung (nur Nullen und Einsen) verarbeitet.

Zahlensysteme

Erinnern Sie sich noch, als in der Schule die verschiedenen Zahlensysteme durchgenommen wurden: das uns so vertraute aus Indien stammende Zehnersystem, das babylonische Sexagesimalsystem und das künstlich anmutende Dual- oder *Binärsystem*? Ich für meinen Teil fand es ebenso interessant zu erfahren, dass die Einteilung unserer Stunden in 60 statt 100 Minuten auf die Babylonier zurückgeht, wie es mich langweilte, Zahlen ins Dualsystem umzurechnen und als Folge von Nullen und Einsen darzustellen. Wer ist denn so dumm, freiwillig mit Binärzahlen zu rechnen? Nun, ich wünschte meine Lehrer hätten mich gewarnt, aber vermutlich wussten die Lehrer damals selbst noch nicht, was man alles mit Binärzahlen anfangen kann (vielleicht haben die Lehrer uns ja auch gewarnt, und wir haben es nur verschlafen). Jedenfalls rechnen Computer nur im Binärsystem:

- zum einem weil die beiden einzigen möglichen Werte 0 und 1 sich gut mit elektronischen Signalen darstellen lassen (Strom an, Strom aus),

- zum anderen weil es sich im Binärsystem sehr leicht rechnen lässt, vorausgesetzt man stößt sich nicht an der kryptischen Darstellung der Zahlen.

Damit wären wir wieder beim eigentlichen Thema. Für den Computer sind also sämtliche Daten (nicht nur die Zahlen) Folgen von Nullen und Einsen, weil er diese am schnellsten und einfachsten verarbeiten kann. Wie diese Daten und die Ergebnisse seiner Berechnungen zu interpretieren sind, ist dabei nicht sein Problem – es ist unser Problem.

Datentypen machen das Leben leichter

Stellen Sie sich vor, wir schreiben das Jahr 1960 und Sie sind stolzer Besitzer einer Rechenmaschine, die Zahlen und Text verarbeiten kann. Beides allerdings in Binärformat. Um Ihre Freunde zu beeindrucken, lassen Sie den »Computer« eine kleine Subtraktion berechnen, sagen wir:

```
8754 – 398 = ?
```

Zuerst rechnen Sie die beiden Zahlen durch fortgesetzte Division durch 2 ins Binärsystem um (wobei die nicht teilbaren Reste der aufeinanderfolgenden Divisionen, von rechts nach links geschrieben, die gewünschte Binärzahl ergeben).

```
10001000110010 – 110001110 = ?
```

Die Binärzahlen stanzen Sie sodann als Lochkarte und lassen diese von Ihrem Computer einlesen. Dann drücken Sie noch die Taste für Subtraktion und ohne Verzögerung erscheint das korrekte Ergebnis:

```
10000010100100
```

Zweifelsohne werden Ihre Freunde von dieser Maschine äußerst beeindruckt sein, und ich selbst wünschte, ich hätte im Mathematik-Unterricht eine derartige praktische Hilfe gehabt. Trotzdem lässt sich nicht leugnen, dass die Interpretation der Binärzahlen etwas unhandlich ist, und zwar erst recht, wenn man neben einfachen ganzen Zahlen auch Fließkommazahlen, Texte und Bitmaps in Binärformat speichert.

Für die Anwender von Computern ist dies natürlich nicht zumutbar und die Computer-Revolution – die vierte der großen Revolutionen (nach der Glorious Revolution, England 1688, der französischen Revolution von 1789 und der Oktoberrevolution, 1917 in Russland) – hätte nicht stattgefunden, hätte man nicht einen Ausweg gefunden. Dieser bestand nun einfach darin, es der Software – dem laufenden Programm – zu überlassen, die vom Anwender eingegebenen Daten (seien es Zahlen, Text, Bitmaps etc.) in Binärformat umzuwandeln und umgekehrt die auszugebenden Daten wieder vom Binärformat in eine leicht lesbare Form zu verwandeln.

»Gemeinheit«, höre ich Sie aufbegehren, »da wurde das Problem ja nur vom Anwender auf den Programmierer abgewälzt«. Ganz so schlimm ist es nicht. Der Java-Compiler nimmt uns hier das Gröbste ab. Alles, was wir zu tun haben, ist, dem Compiler anzugeben, mit welchen Daten wir arbeiten möchten und welchem Datentyp diese Daten angehören (sprich, ob es sich um Zahlen, Text oder Sonstiges handelt).

Schauen wir uns gleich mal ein Beispiel an:

```
public class CErstesBeispiel {
  public static void main(String[] args) {
    int ersteZahl;
    int zweiteZahl;
    int ergebnis;

    ersteZahl = 8754;
    zweiteZahl = 398;
    System.out.println(" 1. Zahl  = " + ersteZahl);
    System.out.println(" 2. Zahl  = " + zweiteZahl);
  }
}
```

Das Grundgerüst, das bereits in Abschnitt 2.2 vorgestellt wurde, übernehmen wir einfach wie gehabt. Wenden wir unsere Aufmerksamkeit gleich den Vorgängen in der `main()`-Funktion zu.

Dort werden zuerst die für die Berechnung benötigten Variablen deklariert.

Die Variablen eines Programms sind nicht mit den Variablen mathematischer Berechnungen gleichzusetzen. *Variablen* bezeichnen Speicherbereiche im RAM (Arbeitsspeicher), in denen ein Programm Werte ablegen kann. Um also mit Daten arbeiten zu können, müssen Sie zuerst eine Variable für diese Daten deklarieren. Der Compiler sorgt dann dafür, dass bei Ausführung des Programms Arbeitsspeicher für die Variable reserviert wird. Für den Compiler ist der Variablenname einfach ein Verweis auf den Anfang eines Speicherbereichs. Als Programmierer identifiziert man eine Variable mehr mit dem Wert, der gerade in dem zugehörigen Speicherbereich abgelegt ist.

Bei der *Deklaration* geben Sie nicht nur den Namen der Variablen an, sondern auch deren Datentyp. Dieser Datentyp gibt dem Compiler an, wie der Inhalt des Speicherbereichs der Variablen zu interpretieren ist. Im obigen Beispiel benutzen wir nur den Datentyp `int`, der für einfache Ganzzahlen steht.

Zu jeder Variablendeklaration gehört auch die Angabe eines *Datentyps*. Dieser gibt dem Compiler an, wie der Speicherinhalt der Variablen zu interpretieren ist.

MERKSATZ

```
int ersteZahl;
```

Dank des Datentyps können wir der Variablen `ersteZahl` direkt eine Ganzzahl zuweisen und brauchen nicht wie im obigen Beispiel des Lochkartenrechners die Dezimalzahl in Binärcode umzurechnen:

```
ersteZahl = 8754;
```

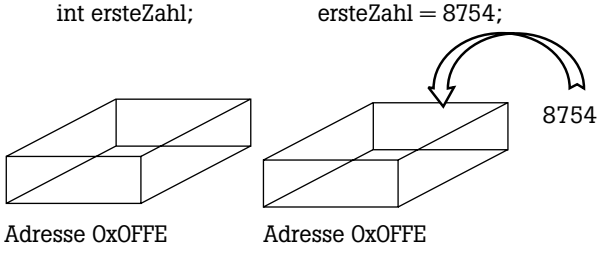

Abb. 3.1: Deklaration und Zuweisung

Der »Wert« der Variablen

Wenn eine Variable einen Speicherbereich bezeichnet, dann ist der Wert einer Variablen der interpretierte Inhalt des Speicherbereichs. Im obigen Beispiel wäre der Wert der Variablen `ersteZahl` nach der Anweisung

```
ersteZahl = 8754;
```

also 8754. Wenn Sie der Variablen danach einen anderen Wert zuweisen würden, beispielsweise

```
ersteZahl = 5;
```

wäre der Wert in der Folge gleich 5.

Was die Variablen für den Programmierer aber so wertvoll macht, ist, dass er sich nicht mehr um die Speicherverwaltung zu kümmern braucht. Es ist zwar von Vorteil, wenn man weiß, dass hinter einer Variablen ein Speicherbereich steht, für die tägliche Programmierarbeit ist es aber meist nicht erforderlich. Wir sprechen nicht davon, dass wir mithilfe des Variablennamens einen eindeutig bezeichneten Platz im Arbeitsspeicher referenzieren und in diesen einen Wert schreiben, wir sagen einfach, dass wir der Variablen einen Wert zuweisen. Wir sprechen nicht davon, dass das interpretierte Bitmuster in dem Speicherbereich der `int`-Variablen `ersteZahl` gleich 5 ist, wir sagen einfach, `ersteZahl` ist gleich 5. Wir sprechen nicht davon, dass wir mithilfe des Variablennamens einen eindeutig bezeichneten Platz im Arbeitsspeicher referenzieren und dessen Wert auslesen, wir sagen einfach, dass wir den Wert der Variablen auslesen.

Mit Variablen arbeiten

Fassen wir noch einmal die drei wichtigsten Schritte bei der Arbeit mit Variablen zusammen:

1. *Variablen müssen deklariert werden*. Die Deklaration teilt dem Compiler nicht nur mit, wie der Speicherbereich für die Variable eingerichtet werden soll, sie zeigt dem Compiler überhaupt erst an, dass es sich bei dem von Ihnen gewählten Namen um einen Variablennamen handelt.
2. *Variablen werden initialisiert*. Als Initialisierung bezeichnet man die anfängliche Zuweisung eines Werts an eine Variable. Die Initialisierung erfolgt meist im Zuge der Deklaration oder kurz danach, um zu verhindern, dass man den Wert einer Variablen ausliest, der zuvor kein vernünftiger Wert zugewiesen wurde.
3. *Variablen werden benutzt*, d.h., ihre Werte werden in Anweisungen ausgelesen oder neu gesetzt.

Listing 3.1: CErstesBeispiel.java

```
public class CErstesBeispiel {
  public static void main(String[] args) {
    int ersteZahl;                          // Deklaration
    int zweiteZahl;
    int ergebnis;

    ersteZahl = 8754;                       // Initialisierung
    zweiteZahl = 398;

    ergebnis = ersteZahl - zweiteZahl;      // Verwendung
    System.out.println(" 8754 - 398  = " + ergebnis);
  }
}
```

Das Wunder der Deklaration

Zum Teufel mit diesen Wortspielen! Soll das jetzt bedeuten, dass die Deklaration einer Variablen ihrer Geburt gleichkommt?

Genau das!

C-Programmierer werden jetzt ins Grübeln kommen. Sollte man nicht zwischen *Deklaration und Definition* unterscheiden, und wenn ja, wäre dann nicht eher die Definition der Variablen mit ihrer Geburt zu vergleichen? Schon richtig, aber in Java wird nicht mehr zwischen Deklaration und Definition unterschieden.

In C bezeichnete man als Deklaration die Einführung des Variablennamens zusammen mit der Bekanntgabe des zugehörigen Datentyps. Die Reservierung des Speichers und die Verbindung des Speichers mit der Variablen erfolgten aber erst in einem zweiten Schritt, der sogenannten Definition.

Allerdings ist die Unterscheidung etwas verwischt, denn die Deklaration einfacher Variablen schließt meist deren Definition ein.

In Java schließlich ist die Variablendeklaration immer mit einer Speicherreservierung verbunden.

Jetzt wissen wir also, wozu Variablen deklariert werden, wir wissen, welche Vorgänge mit der Deklaration verbunden sind, und wir wissen, dass die Deklaration immer der Benutzung der Variablen vorangehen muss, da der Compiler ja sonst nichts mit dem Variablennamen anfangen kann. Was wir nicht wissen, ist, was es genau heißt, wenn wir so salopp sagen, »die Deklaration muss der Benutzung *vorangehen*.« Um nicht schon wieder vorgreifen zu müssen, verweisen wir diesmal auf die weiter unten folgenden Abschnitte 3.4, »Methoden von Klassen«, und 5.3, »Dreierlei Variablen«, wo wir diese Frage klären werden. Im Moment, da wir uns nur mit sogenannten lokalen Variablen beschäftigen, die innerhalb einer Methode deklariert werden (die anderen Variablentypen hängen mit der Definition von Klassen zusammen und werden später beschrieben), begnügen wir uns mit dem Hinweis, dass die Deklaration der Variablen vor, d.h. im Quelltext über, der Benutzung der Variablen stehen muss. Am übersichtlichsten ist es, die Deklarationen gebündelt an den Anfang der Methode zu stellen.

Die einfachen Datentypen

Nun aber wieder zurück zu Variablen und Datentypen. Außer dem Datentyp `int` für Ganzzahlen kennt Java noch eine Reihe weiterer einfacher Datentypen:

Tabelle 3.1: Einfache Datentypen

Datentyp	Beschreibung	Wertebereich
`boolean`	boolescher Wert (wahr, falsch)	`true`, `false`
`char`	Zeichen, Buchstabe	Unicode-Werte
`byte`	ganze Zahl	–128 bis +127
`short`	ganze Zahl	–32768 bis 32767
`int`	ganze Zahl	–2.147.483.648 bis +2.147.483.647
`long`	ganze Zahl	–9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
`float`	Fließkommazahl	$-3{,}40282347*10^{38}$ bis $+3{,}40282347*10^{38}$
`double`	Fließkommazahl	$-1{,}7976931348623157*10^{308}$ bis $+1{,}7976931348623157*10^{308}$

Unicode In der Tabelle ist für `char`-Variablen der Unicode angegeben. Was sich recht unscheinbar anhört, ist eine bahnbrechende Neuerung! *Unicode* ist ein standardisierter Zeichensatz mit über 90.000 Zeichen, mit dem alle diversen Umlaute und Sonderzeichen aller gängigen Sprachen, ja sogar japanische und chinesische Schriftzeichen dargestellt werden können!

Wie Sie sehen, gibt es verschiedene Datentypen mit unterschiedlichen Wertebereichen. Um z.B. eine ganze Zahl abzuspeichern, haben Sie die Wahl zwischen `byte`, `short`, `int` und `long`! Die größeren Wertebereiche erkauft man sich mit einem höheren Speicherverbrauch. Eine `long`-Variable benötigt beispielsweise doppelt so viel Speicher wie eine `int`-Variable. Glücklicherweise ist Arbeitsspeicher kein allzu großes Problem mehr und viele Programmierer verwenden standardmäßig `long` für ganzzahlige Werte und `double` für Fließkommazahlen.

Der *Datentyp* legt also nicht nur fest, wie der Wert der Variablen zu interpretieren ist, er gibt auch an, wie groß der für die Variable bereitzustellende Speicherbereich sein muss.

Schauen wir uns einige Beispiele an:

```
int ganzeZahl;
double krummeZahl;
boolean ja, nein, oder_doch;
boolean Antwort;
short klein = -4;
char buchstabe;
char Ziffer;

ganzeZahl = 3444;
krummeZahl = 47.11;
buchstabe = 'Ü';
Ziffer = '4';
Antwort = true;
```

Wie Sie an den Beispielen sehen, kann man auch mehrere Variablen des gleichen Typs durch Komma getrennt auf einmal deklarieren und es ist sogar erlaubt, eine Variable direkt im Zuge ihrer Deklaration zu initialisieren, d.h. ihr einen ersten Wert zuzuweisen (siehe `klein`).

Das hört sich ganz so an, als sei der Java-Compiler, der Ihren Quelltext in binären Bytecode übersetzt, recht großzügig, was die verwendete Syntax angeht. Nun, dem ist keineswegs so.

Java für Pedanten

Auch wenn Ihnen die Syntax von Java einerseits viele Möglichkeiten offen lässt, ist sie andererseits doch recht starr vorgegeben und der Compiler wacht penibel darüber, dass Sie sich an die korrekte Syntax halten.

Wenn Sie es sich also nicht mit dem pedantischen Compiler verderben wollen, sollten Sie insbesondere auf folgende Punkte achten:

- Alle *Anweisungen* (also Zuweisungen, Funktionsaufrufe und Deklarationen) müssen mit einem Semikolon abgeschlossen werden.

```
krummeZahl = 47.11;
```

- Java unterscheidet streng zwischen *Groß- und Kleinschreibung*. Wenn Sie also eine Variable namens `krummeZahl` deklariert haben, dann müssen Sie auch `krummeZahl` schreiben, wenn Sie auf die Variable zugreifen wollen, und nicht `krummezahl`, `KrummeZahl` oder `KRUMMEZAHL`.

Und natürlich gibt es auch spezielle Regeln für die Auswahl von Bezeichnern (Namen von Variablen, Methoden, Klassen).

- *Bezeichner* können beliebig lang sein, müssen mit einem Buchstaben,[1] '_' oder '$' beginnen und dürfen nicht identisch zu einem Schlüsselwort der Sprache sein.

Sie dürfen Ihre Variable also nicht `class` nennen, da dies ein reserviertes Schlüsselwort der Sprache ist. Im Anhang finden Sie eine Liste der reservierten Wörter.

Nicht obligatorisch, aber allgemein üblich ist es, in Java Variablennamen grundsätzlich klein zu schreiben. In Namen, die aus mehreren Teilen zusammengesetzt sind, beginnt jeder Teil mit einem Großbuchstaben. Der erste Buchstabe ist aber immer klein:

```
eineVariable, zahl, nochEineZahl, alter
```

Variablen versus Konstanten

Muss man wirklich erst erwähnen, dass man den Wert einer Variablen ändern kann, indem man ihr einen neuen Wert zuweist (d.h. einen neuen Wert in ihren Speicherbereich schreibt), während der Wert einer Konstan-

1. Als »Buchstaben« gelten unter anderem auch die deutschen Umlaute. Wenn Sie also eine Variable `begrüßung` nennen wollen, brauchen Sie sie nicht wie in anderen Programmiersprachen als `begruessung` zu deklarieren, sondern können ruhig `begrüßung` schreiben. In Klassen- und Dateinamen sollten Sie allerdings keine Umlaute verwenden, da dies auf manchen Plattformen zu Schwierigkeiten bei der Kompilation führen kann.

ten unverändert bleibt? Wohl nicht. Interessanter ist es schon zu erfahren, wie man mit Konstanten arbeitet. Dazu gibt es zwei Möglichkeiten:

Erstens:

Sie tippen die Konstante direkt als Wert ein, man spricht dann von sogenannten *Literalen*.

```
krummeZahl = 47.11;                    // Zuweisung eines Literals
krummeZahl = ganzeZahl + 47.11;
```

Da mit einem Literal kein Datentyp verbunden ist, muss der Compiler den Datentyp aus der Syntax des Literals ablesen:

Tabelle 3.2: Literale

Datentyp	Literal
`boolean`	`true, false`

`char`	`'c', 'Ü'  // einfaches Zeichen`
	`'\n', '\\',  // Sonderzeichen`
	`'\u1234'  // Unicode-Codierung`

`String`	`"Dies ist ein String"`

`int`	`12, -128`
	`077  // oktal`
	`0xFF1F  // hexadezimal`

`long`	`12L, 1400000`
`float`	`12.4f, 10e-2f`
`double`	`47.11, 1e5`

Zweitens:

Sie deklarieren eine Variable mit dem Schlüsselwort `final`:

```
final double KRUMMEZAHL = 47.11;
final double PI = 3.141592654;
```

Der Wert einer solchen »konstanten Variablen« kann nach der Initialisierung (ersten Wertzuweisung) nicht mehr verändert werden.

Konstante Variablen werden gemäß allgemeiner Konvention ganz in Großbuchstaben geschrieben.

3.2 Operatoren

Nachdem wir nun gesehen haben, was Variablen sind und wie man sie definiert und ihnen einen Wert zuweist, sollten wir nun endlich auch damit beginnen, etwas Sinnvolles mit ihnen zu machen. Dazu dienen bei den bisher vorgestellten einfachen Datentypen vor allem die sogenannten Operatoren.

Listing 3.2: COperatoren.java

```
public class COperatoren {
  public static void main(String[] args) {
    int x,y,z;
    int ergebnis_1,ergebnis_2;

    x = 1;
    y = 2;
    z = 3;

    ergebnis_1 = x + y * z;             // = 7
    ergebnis_2 = (5 - 3) * z;           // = 6
    System.out.println(ergebnis_1);
    System.out.println(ergebnis_2);

    x = x + z;                          // = 4
    System.out.println(x);
    x += z;                             // = 7
    System.out.println(x);
    x += 1;                             // = 8
    System.out.println(x);
    x++;                                // = 9
    System.out.println(x);
  }
}
```

Das schaut doch ziemlich vertraut aus, oder? Eigentlich genau so, wie man es von algebraischen Gleichungen her kennt. Aber achten Sie bitte auf die letzten Zeilen des Beispiels. Hier sehen wir seltsame Konstruktionen, die wir nun erklären wollen:

```
x = x + z;
```

Diese Anweisung bewirkt, dass der Computer die aktuellen Werte von x und z zusammenaddiert und dann in x speichert, d.h., die Variable x enthält nach Ausführung dieser Zeile als neuen Wert die Summe aus ihrem alten Wert und z.

Da das Hinzuaddieren eines Werts zum Wert einer Variablen sehr häufig vorkommt, gibt es dafür eine Kurzschreibweise, nämlich:

```
x += z;
```

Dies teilt dem Rechner mit, dass er zum Wert von x den Inhalt von z hinzuaddieren und das Ergebnis wieder in x speichern soll.

Sehr oft möchte man eine Variable hochzählen (*inkrementieren*). Java kennt auch hierfür einen speziellen Operator: ++.

```
x++;
```

Diese Anweisung erhöht den Wert von x um 1. Äquivalente Anweisungen wären:

`x = x + 1;` oder `x += 1;`

Aber Programmierer sind schreibfaul und x++ sieht ja auch viel geheimnisvoller aus!

Das oben Gesagte gilt gleichermaßen für die anderen Grundrechenarten (–, *, /) und das Dekrementieren von Variablen (--).

Die verschiedenen Operatoren

In Java gibt es natürlich noch andere Operatoren. Die wichtigsten sind:

Tabelle 3.3: Operatoren

Operator	Beschreibung	Beispiel
++, --	Inkrement, Dekrement	Erhöht oder erniedrigt den Wert einer Variablen um 1.
!	logisches NICHT	Negiert den Wahrheitswert einer Aussage (beispielsweise eines Vergleichs). Wird meist in Kontrollstrukturen (siehe Kapitel 4) verwendet.
*, /	Multiplikation, Division	Multiplikation und Division
%	Modulo-Division	Liefert den Rest einer ganzzahligen Division. 4 % 3 liefert z.B. 1.
–, +	Subtraktion, Addition	Subtraktion und Addition
<=, <, >, >=	Vergleich	Zum Vergleich zweier Werte. Die Operatoren liefern `true` oder `false` zurück.
==, !=	Vergleich (gleich, ungleich)	Zum Vergleich auf Gleichheit oder Ungleichheit. Die Operatoren liefern `true` oder `false` zurück.

Tabelle 3.3: Operatoren (Forts.)

Operator	Beschreibung	Beispiel
&&	logisches UND	Verknüpft zwei Aussagen. Liefert `true`, wenn beide Aussagen `true` sind. `if ( (x < 1) && (y > 1) )`
||	logisches ODER	Verknüpft zwei Aussagen. Liefert `true`, wenn eine der beiden Aussagen `true` ist. `if ( (x < 1) || (y > 1) )`

&	bitweises UND	UND-Verknüpfung der Binärpräsentation zweier Zahlen.
		`var1 = 1; // ...0001`
		`var2 = 5; // ...0101`
		`var3 = var1 & var2; // ...0001`

|	bitweises ODER	ODER-Verknüpfung der Binärpräsentation zweier Zahlen.
		`var1 = 1; // ...0001`
		`var2 = 5; // ...0101`
		`var3 = var1 | var2; // ...0101`

Die Reihenfolge in der Tabelle deutet die *Priorität* der Operatoren bei der Auswertung von Ausdrücken an. Beispielsweise sind * und / höher eingestuft als + und –, was genau der altbekannten Schulregel entspricht »Punktrechnung vor Strichrechnung«.

Ein *Ausdruck* ist eine Berechnung aus Variablen, Konstanten und Operatoren, die auf der rechten Seite einer Zuweisung steht.

Wenn man sich bei der Reihenfolge nicht ganz sicher ist oder eine bestimmte Reihenfolge der Auswertung erzwingen möchte, kann dies durch die Verwendung von Klammern erreicht werden. Aber auch wenn keine direkte Notwendigkeit zum Setzen von Klammern besteht, können Sie diese verwenden, um eine Berechnung besser lesbar zu machen.

```
z *= ((2*loop)/(2*loop-1)) * ((2*loop)/(2*loop+1));
```

3.3 Typumwandlung

Damit wissen Sie schon fast alles, was ein guter Java-Programmierer über Variablen, Operatoren und einfache Datentypen (nennt man manchmal auch elementare, primitive oder built-in Datentypen) wissen muss.

Aber ein wichtiger Aspekt fehlt noch: Was passiert, wenn Ausdrücke mit verschiedenen Datentypen auftreten? Darf man Datentypen mischen? Die Antwort kommt von Radio Eriwan: Ja, aber ...

Automatische Typumwandlung

Schauen wir zunächst ein Code-Beispiel an.


```
public class CDemo1 {
   public static void main(String[] args) {
      int x = 4711;
      double y;

      y = x;
      System.out.println(y);
   }
}
```

Die Variable y kann nur Fließkommazahlen (`double`) speichern, soll aber einen `int`-Wert zugewiesen bekommen. Ist diese Zuweisung erlaubt? Ja! Die Umformatierung des Integer-Werts 4711 in den Fließkommawert 4711.0 bereitet dem Compiler keine Mühen.

Doch nicht immer geht alles so glatt!


```
public class CDemo2 {
  public static void main(String[] args)  {
    int x;
    double y = 3.14;

    x = y;
    System.out.println(x);
  }
}
```

In diesem Falle soll die Integer-Variable x einen Fließkommawert (`double`) aufnehmen. Ist diese Zuweisung erlaubt? Ja und nein! Wenn Sie obigen Code kompilieren, beschwert sich der Compiler, weil er eine Fließkommazahl in eine Integer-Variable quetschen soll, und dies ist meist mit Datenverlusten verbunden.

Mithilfe einer expliziten Typumwandlung können wir den Compiler aber zwingen, die gewünschte Umformatierung vorzunehmen.

Explizite Typumwandlung (Casting)

Um eine Typumwandlung zu erzwingen, die der Compiler nicht automatisch unterstützt, stellt man einfach dem zu konvertierenden Wert den gewünschten Datentyp in Klammern voran. Im Beispiel Demo2 würden wir also schreiben:

```
x = (int) y;
```

Aber man muss auf der Hut sein. Hier soll eine Bruchzahl in einen ganzzahligen Wert umgewandelt werden. Der Compiler behilft sich in diesem Fall einfach damit, dass er den Nachkommateil wegwirft und x den Wert 3 zuweist. Es gehen also Daten verloren bei der Umwandlung (Neudeutsch *cast*) von `double` zu `int`.

Bereichsüberschreitung

Manchmal merkt man auch gar nicht, dass man den falschen Typ verwendet hat. Dann kann auch der Compiler nicht mehr helfen.


```
public class CDemo3 {
  public static void main(String[] args)  {
    int x,y;
    short z;

    x = 30000;
    y = 30000;

    z = (short) (x + y);
    System.out.println(z);
  }
}
```

Eine böse Falle! `x + y` ergibt 60000 und das ist außerhalb des Wertebereichs von `short`! Das Ergebnis lautet in diesem Fall -5536.

Wie kommt dieses merkwürdige Ergebnis zustande?

Codierung von short

Als Integer-Wert wird 60000 als 32-Bit-Wert codiert:

```
0000 0000 0000 0000 1110 1010 0110 0000
```

Eine `short`-Variable verfügt aber nur über 16 Bit Arbeitsspeicher. Der Compiler schneidet bei der Typumwandlung also erst einmal die obersten 16 Bit weg. Übrig bleibt:

```
1110 1010 0110 0000
```

Dieses Bitmuster wird nun als `short`-Wert interpretiert. Das bedeutet, dass das oberste Bit zur Codierung des Vorzeichens und nur die fünfzehn unteren Bits zur Codierung des Werts benutzt werden.

```
 110 1010 0110 0000 = 27232
```

Nun muss man noch wissen, dass der Compiler die negativen Zahlen von unten nach oben quasi rückwärts zählt, wobei die größte, nicht mehr darstellbare negative Zahl 32768 ist.

-32768 + 27232 = -5536. Voilà, da haben wir unseren Wert.

Division

Im nächsten Versuch soll ein einfacher Bruch berechnet werden. So einfach und doch ein Stolperstein für viele Programmierer.


```
public class CDemo4 {
  public static void main(String[] args) {
    int x,y;
    double z1,z2;

    x = 3;
    y = 4;
    z1 = x / y;
    z2 = 3/4;
    System.out.println(z1);
    System.out.println(z2);
  }
}
```

Was glauben Sie, welche Werte z1 und z2 haben? Bestimmt nicht 0,75, wie man leichtfertig annehmen könnte. Beide sind 0! Wie kommt denn das?

Nun, denken Sie an die pedantische Vorgehensweise des Compilers. Er wertet die Ausdrücke Schritt für Schritt und streng nach Vorschrift aus.

Bei `z1 = 3/4;` wird zunächst die Division 3/4 ausgeführt. Da beide beteiligten Operanden ganzzahlig sind, wird nach einer »internen Dienstanweisung« auch das Ergebnis 0.75 in einen ganzzahligen Wert konvertiert, d.h., der Nachkommateil fällt weg und es bleibt eine Null übrig. Nun erst erfolgt die Zuweisung an die `double`-Variable z1. Pflichtbewusst wird daher die `int`-0 in eine `double`-0.0 konvertiert und an z1 zugewiesen. Analoges passiert bei `z2 = x/y`.

Was kann man nun tun, um das gewünschte Ergebnis zu erhalten?

Eine weitere »interne Dienstvorschrift« sagt dem Compiler, dass alle Operanden eines Ausdrucks den gleichen Datentyp haben müssen, und zwar den »größten«, der auftaucht. Es reicht also, wenn wir einen Operanden explizit umwandeln lassen:

```
z1 = (double) x / y;
z2 = (double) 3/4;
```

Das Voranstellen des gewünschten Datentyps in Klammern veranlasst den Compiler, aus der ganzzahligen 3 eine `double`-3.0 zu machen. Dadurch

greift beim nachfolgenden Auswerten der Division die besagte Regel, dass alle Operanden den größten auftretenden Typ haben müssen. Der Compiler castet daher auch die 4 zu 4.0 und wir haben eine reine `double`-Division `3.0 / 4.0` vorliegen. Das Ergebnis ist daher auch ein `double`-Wert und `z1` und `z2` erhalten beide den korrekten Wert 0.75.

Bei Zahlenkonstanten wie 3/4 kann man auch gleich eine `double`-Zahl schreiben, also `z1 = 3.0/4.0;`

Sie haben aber wohl schon gemerkt, dass man sehr leicht Fehler einbauen kann, besonders bei etwas größeren Programmen oder langen Formeln, die berechnet werden sollen. Daher unser Tipp:

Verwenden Sie nach Möglichkeit bei Berechnungen immer nur einen einzigen Datentyp, vorzugsweise `double`. Alle beteiligten Variablen sollten diesen Typ haben und auftretende Zahlenkonstanten immer in Dezimalschreibweise (also 47.0, 1.0 usw.) schreiben. Sie werden sich dadurch manche Fehlersuche ersparen! Wenn Sie viele Berechnungen durchführen, kann allerdings u.U. der Datentyp `float` zur schnelleren Abarbeitung führen (da dieser Datentyp am besten zu den üblichen 32-Bit Prozessoren passt).

3.4 Objekte und Klassen

Wie schon mehrfach angeklungen ist, existieren neben den beschriebenen elementaren Datentypen noch komplexere und das sind diese seltsamen Teile, die wir nun schon mehrere Male angetroffen, aber meist mehr oder weniger ignoriert haben: die Klassen.

Java für Philosophen

Bevor wir uns konkret anschauen, wie man eigene Klassen erstellt und bereits vordefinierte Klassen in seinen Programmen verwendet, wollen wir einen kurzen Blick auf die Philosophie werfen, die hinter dem Schlagwort *Objektorientierung* steckt, denn OOP (objektorientierte Programmierung) steht mehr für eine spezielle Sichtweise als eine ganz neue Programmiertechnik.

Zäumen wir das Pferd von hinten auf und stellen wir uns zunächst die Frage: Wie sieht denn die nicht objektorientierte Programmierung aus?

Nun, man definiert die notwendigen Variablen ähnlich wie in den kleinen Beispielen von vorhin, und dann setzt man die Anweisungen auf, die mit diesen Variablen arbeiten. Fast alle Programmiersprachen bieten dabei die Möglichkeit, Anweisungen in sogenannten Funktionen zu bündeln und aus-

zulagern. Der Programmierer hat dann die Möglichkeit, seinen Code in mehrere Funktionen aufzuteilen, die jede eine bestimmte Aufgabe erfüllen (beispielsweise das Einlesen von Daten aus einer Datei, die Berechnung einer mathematischen Funktion, die Ausgabe des Ergebnisses auf dem Bildschirm). Damit diese Funktionen zusammenarbeiten können, tauschen sie auf verschiedenen Wegen Variablen und Variablenwerte aus.

Bei diesem Modell haben wir auf der einen Seite die Daten (abgespeichert in Variablen) und auf der anderen Seite die Funktionen, die mit Daten arbeiten. Dabei sind beide Seiten prinzipiell vollkommen unabhängig voneinander. Welche Beziehung zwischen den einzelnen Funktionen einerseits und den Funktionen und den Daten andererseits besteht, wird erst klar, wenn man versucht nachzuvollziehen, wie die Funktionen bei Ausführung des Programms Daten austauschen.

Die Erfahrungen mit diesem Modell haben gezeigt, dass bei Programmprojekten, die etwas größer werden, sich sehr leicht Fehler einschleichen: da verändert eine Funktion A nebenbei eine Variable, die später eine Funktion B an ganz anderer Stelle im Programm zum Absturz bringt. Die Fehlersuche dauert dann entsprechend lange, weil die Zusammenarbeit von Daten und Funktionen kaum nachzuvollziehen ist! Ferner tendieren solche Programme dazu, sehr chaotisch zu sein. Eine Wartung (Modifizierung, Erweiterung) zu einem späteren Zeitpunkt ist oft ein kühnes Unterfangen, vor allem, wenn es nicht mehr derselbe Programmierer ist, der nun verzweifelt zwischen Hunderten von Funktionen herumirrt und versucht, die Zusammenhänge und Wirkungsweise zu verstehen.

Schlaue Köpfe kamen daher auf die Idee, eine ganz andere Sichtweise anzunehmen und diese in der Programmiersprache umzusetzen. Ausgangspunkt war dabei die Vorstellung, dass bestimmte Daten und die Funktionen, die mit diesen Daten arbeiten, untrennbar zusammengehören. Eine solche Einheit von logisch zusammengehörigen Daten und Funktionen bildet ein Objekt. Abstrakt formuliert beschreiben die Daten (Variablen) dabei die Eigenschaften des Objekts, und die Funktionen (die dann meist Methoden heißen) legen sein Verhalten fest. Der Datentyp, der die gleichzeitige Deklaration von Datenelementen und Methoden erlaubt, ist die Klasse, angezeigt durch das Schlüsselwort `class`.

Objekte und alte Datentypen

Im Grunde ist dies gar nicht so neu. Denken Sie nur an die einfachen Datentypen und die Operatoren. Stellen Sie sich eine `int`-Variable einfach als ein Objekt mit einem einzigen Datenelement, eben der `int`-Variablen, vor. Die Funktionen, die mit diesem Objekt verbunden sind, sind dann die Operatoren, die auf `int`-Variablen angewendet werden können (Addition, Subtraktion, Vergleiche etc.). Der Vorteil der Klassen liegt allerdings darin, dass in einem Datentyp mehrere Datenelemente vereinigt werden können und dass Sie in Form der Methoden der Klasse selbst festlegen können, welche Operationen auf den Variablen der Klasse erlaubt sind.

Klassen deklarieren

Der erste Schritt bei der objektorientierten Programmerstellung ist die Zerlegung des Problems in geeignete Objekte und die Festlegung der Eigenschaften und Verhaltensweisen, sprich der Datenelemente und der Methoden, die diese Objekte haben sollten. Dies ist z.T. sehr schwierig und braucht oft viel Erfahrung, damit durch sinnvolles Bestimmen der Programmobjekte auch die Vorteile der Objektorientiertheit zum Tragen kommen können!

Überlegen wir uns gleich mal eine kleine Aufgabe. Angenommen, Sie sollen für Ihre Firma ein Programm zur Verwaltung der Mitarbeiter schreiben. Wie könnte eine Aufteilung in Objekte aussehen? Welche Eigenschaften und Methoden sind erforderlich?

Das Schlüsselwort class

Eine naheliegende Lösung ist, die Mitarbeiter als die Objekte anzusehen. Schaffen wir uns also den Prototyp eines Mitarbeiters und implementieren wir diesen in Form der Klasse `CMitarbeiter`.

```
class CMitarbeiter {
}
```


Unter Java-Programmierern ist es üblich, Klassennamen mit einem Großbuchstaben beginnen zu lassen, danach wird klein weiter geschrieben. In zusammengesetzten Namen beginnt jeder Teil mit einem Großbuchstaben. Eine von uns in diesem Buch verwendete Konvention ist, alle selbst definierten Klassen mit einem `C` (für class) beginnen zu lassen. Dadurch lassen sich eigene Klassen direkt von den Java-Klassen aus den Standardbibliotheken unterscheiden.

Wir haben gerade eine Klasse kreiert! War doch gar nicht schwer, oder? Nun müssen wir unserer Klasse noch Eigenschaften und Methoden zuweisen.

Eigenschaften von Klassen

Was brauchen wir, um einen Mitarbeiter zu beschreiben? Na klar, einen Namen und Vornamen wird er wohl haben. Und ein Gehalt kriegt er fürs fleißige Werkeln. Erweitern wir also die Klasse um diese Eigenschaften in Form von geeigneten Variablen:

```
class CMitarbeiter {
  String m_name;
  String m_vorname;
  int    m_gehalt;
}
```

Langsam nimmt unser Objekt konkrete Formen an! Den Datentyp `int` kennen Sie ja schon. `String` ist kein einfacher Datentyp (daher haben wir ihn im vorherigen Abschnitt auch nicht kennengelernt), sondern ebenfalls ein Objekt, eine Klasse also, genau wie unser `CMitarbeiter`. Im Gegensatz zu unserer Klasse ist `String` schon von anderen Leuten erstellt worden (genau gesagt von den Programmierern der Firma Sun) und wird jedem Java-Entwicklungspaket zusammen mit Hunderten anderer nützlicher Klassen mitgegeben.

Aber auf diesen Punkt kommen wir in Kürze ausführlicher zu sprechen. Merken Sie sich im Moment, dass `String` eine Klasse ist und dazu dient, Zeichenketten (englisch Strings) aufzunehmen und zu verarbeiten.

Felder Diese Variablen, die innerhalb einer Klasse, aber außerhalb aller Methoden der Klasse deklariert werden, nennt man *Felder* oder *Membervariablen*. Alle Methoden der Klasse können auf diese Variablen zugreifen.

Damit Sie schnell und sicher erkennen können, ob es sich bei einem Bezeichner um ein Feld oder eine lokal in einer Methode definierte Variable handelt, werden wir für alle Felder Namen verwenden, die mit `m` (für Membervariable[1]) beginnen. Später, wenn Sie etwas erfahrener in der objektorientierten Programmierung sind und eigene Programme schreiben, werden Sie auf dieses Präfix vermutlich verzichten.

Machen wir nun weiter mit dem Ausbau unserer eigenen Klasse. Nehmen wir an, dass Ihr Chef von Ihrem Programm erwartet, dass es folgende Dinge kann:

- die persönlichen Daten eines Mitarbeiters ausgeben
- sein Gehalt erhöhen

Sie scheinen einen netten Chef zu haben! Auf den Gedanken, das Gehalt zu senken, kommt er gar nicht. Lassen wir ihm keine Chance, es sich anders zu überlegen, und versuchen wir, seinen Anforderungen zu entsprechen.

Methoden von Klassen

Beachten Sie bitte, dass *persönliche Daten ausgeben* und *Gehalt erhöhen* Aktionen sind, die auf den Daten des Mitarbeiters operieren. Folglich werden diese als Methoden der Klasse `CMitarbeiter` implementiert:


```
class CMitarbeiter {
  String m_name;
  String m_vorname;
  int m_gehalt;
```

1. Wir haben uns für das `m` als Präfix entschlossen, weil es im Schriftbild etwas unaufdringlicher ist als das `f` für Feld.

```
  CMitarbeiter(String name, String vorname,
               int gehalt) {
    m_name = name;
    m_vorname = vorname;
    m_gehalt = gehalt;
  }

  void datenAusgeben() {
    System.out.println("\n");
    System.out.println(" Name    : " + m_name);
    System.out.println(" Vorname : " + m_vorname);
    System.out.println(" Gehalt  : " + m_gehalt + " Euro");
  }

  void gehaltErhoehen(int erhoehung) {
    m_gehalt += erhoehung;
  }
} //Ende der Klassendeklaration
```

Die Klasse `CMitarbeiter` besitzt nun drei Methoden mit den Namen `CMitarbeiter`, `datenAusgeben` und `gehaltErhoehen`.

Übung

Bevor wir uns diese drei Methoden im Einzelnen anschauen, sollten wir uns überlegen, wie eine Methodendeklaration im Allgemeinen auszusehen hat. Stellen Sie sich vor, dass Sie selbst gerade dabei sind, eine Programmiersprache wie Java zu entwickeln, und stellen Sie zusammen, was für die Deklaration einer Methode erforderlich ist:

Lösung

1. Zuerst braucht die Methode einen Namen, damit sie später aufgerufen werden kann. Wie bei den Variablennamen verbirgt sich hinter dem Methodennamen eine Adresse. Diese weist bei den Methoden allerdings nicht auf einen Speicherbereich, in dem ein Wert abgelegt ist, sondern auf den Code der Methode. (Tatsächlich werden beim Aufruf eines Programms ja nicht nur die Daten in den Arbeitsspeicher kopiert, auch der Programmcode, die auszuführenden Maschinenbefehle, wird in den Speicher geladen.)

   Wird eine Methode aufgerufen, sorgt der Compiler dafür, dass der Code der Methode ausgeführt wird. Nach der Abarbeitung der Anweisungen der Methode wird das Programm hinter dem Aufruf der Methode weitergeführt. Damit hätten wir auch schon den zweiten wichtigen Bestandteil unserer Methodendeklaration:

2. Die Anweisungen, die bei Aufruf der Methode ausgeführt werden sollen. Denken Sie dabei daran, dass zusammengehörende Anweisungsblöcke in geschweifte Klammern gefasst werden.

3. Letztlich sollte der Compiler schnell erkennen können, dass ein Name eine Methode bezeichnet. Vereinbaren wir daher einfach, dass auf den Methodennamen zwei Klammern folgen sollen.

Unsere Methodendeklaration sieht damit folgendermaßen aus:

```
methodenName(){
   Anweisungen;
}
```


Mittlerweile haben wir die dritte Art von *Bezeichnern* (Namen, die der Programmierer einführt und per Deklaration dem Compiler bekannt gibt) kennengelernt. Die erste Art von Bezeichnern waren die Variablennamen, die zweite Art von Bezeichnern stellen die Namen dar, die wir den selbst definierten Klassen geben, und die dritte Art von Bezeichnern sind die Methodennamen.

Woher nimmt die Methode die Daten, mit denen sie arbeitet?

- Nun, zum einem ist eine Methode ja Bestandteil einer Klassendeklaration. Für die Methode bedeutet dies, dass sie auf alle *Felder* ihrer Klasse zugreifen kann. (Zur Erinnerung: Dies sind die Variablen, die innerhalb der Klasse, aber außerhalb jeder Methode deklariert sind.)
- Zum anderen kann eine Methode natürlich auch eigene, sogenannte *lokale Variablen* definieren. Von diesen haben wir in den vorangegangenen Abschnitten bereits eifrig Gebrauch gemacht. Alle dort deklarierten Variablen waren lokale Variablen der Methode `main()`. Diese lokalen Variablen sind keine Klassenelemente, folglich können sie nicht in jeder beliebigen Methode der Klasse benutzt werden, sondern nur innerhalb der Methode, in der sie deklariert sind.

Wie aber, wenn zwei Methoden unterschiedlicher Klassen Daten austauschen sollen?

4. Für den Austausch über Klassengrenzen hinweg sehen wir sogenannte Parameter vor. Dies sind Variablen, die innerhalb der Klammern der Methodendeklaration deklariert werden. Bei Aufruf der Methode werden diesen Parametern Werte übergeben (die sogenannten Argumente), die dann innerhalb der Methode wie lokale Variablen benutzt werden können.
5. Schließlich soll die Methode auch noch Daten nach außen exportieren. Zu diesem Zweck definiert jede Methode einen Rückgabewert, dessen Datentyp vor den Methodennamen gestellt wird. Später in Abschnitt 5.2 werden wir dann noch sehen, wie mithilfe des Schlüsselwortes `return` dieser Rückgabewert an den Aufrufer der Methode zurückgeliefert wird.

Eine vollständige Methodendeklaration würde jetzt folgendem Schema folgen:

```
Rückgabetyp methodenName(Deklarationen_der_Parameter)  {
  lokaleVariablen;
  Anweisungen;
}
```

Schauen wir uns jetzt die beiden Methoden `datenAusgeben` und `gehaltErhoehen` aus unserem Beispiel an.

```
void datenAusgeben() {
   System.out.println("\n");
   System.out.println(" Name    : " + m_name);
   System.out.println(" Vorname : " + m_vorname);
   System.out.println(" Gehalt  : " + m_gehalt + " Euro");
}
```

Die leeren Klammern `()` besagen, dass keine Parameter übergeben werden. Das `void` zeigt an, dass auch kein Wert zurückgegeben wird. Die Methode `datenAusgeben()` erwartet also weder irgendwelche Parameter noch gibt sie beim Ausführen einen Wert zurück. Schauen wir nun in das Innere der Methode (also was zwischen dem Klammernpaar { } steht).

println()

Dort finden wir wieder die Methode `System.out.println()`, die wir schon die ganze Zeit zur Ausgabe benutzen. Ihr können Sie als Parameter einen auszugebenden Text (eingeschlossen in Hochkommata) oder Variablen der einfachen Datentypen und der Klasse `String` übergeben, deren Inhalt ausgegeben werden soll.

Mehrere in einer Zeile auszugebende Texte und Variablen können Sie mithilfe des +-Operators verbinden.

Das Zeichen `\n` bewirkt bei der Ausgabe einen zusätzlichen Zeilenumbruch und dient hier nur zur optischen Verschönerung.

Gehen wir weiter zur Methode `gehaltErhoehen`.

```
void gehaltErhoehen(int erhoehung)  {
    m_gehalt += erhoehung;
}
```

Das Schlüsselwort `void` gibt wiederum an, dass die Methode keinen Wert an die aufrufende Stelle zurückgibt. In den Klammern finden wir als Parameter eine `int`-Variable namens `Erhoehung`, die im Anweisungsteil zum aktuellen Gehalt addiert wird.

Konstruktoren von Klassen

Nun zu der Methode, die den gleichen Namen trägt wie die ganze Klasse:

```
CMitarbeiter(String name, String vorname, int gehalt) {
    m_name = name;
    m_vorname = vorname;
    m_gehalt = gehalt;
}
```

Dies ist eine ganz besondere Klassenfunktion, nämlich ein *Konstruktor*. Jede Klasse braucht einen oder sogar mehrere Konstruktoren, die beim Initialisieren der Variablen der Klasse behilflich sind. In unserem Fall übergeben wir die persönlichen Daten des Mitarbeiters an den Konstruktor, der sie den richtigen Variablen zuweist. Bitte beachten Sie, dass die Parameter anders heißen als die Felder (schließlich stellen die Parameter eigenständige Variablen dar und müssen daher auch eigene Namen haben).

Jede Klasse braucht zumindest einen Konstruktor zur Initialisierung ihrer Felder. Wenn Sie selbst keinen solchen Konstruktor vorsehen, weist der Compiler der Klasse einen Standardkonstruktor zu.

Damit ist die Mitarbeiter-Klasse fürs Erste vollendet! Das war doch nicht allzu schwer?! Nun wollen wir diese Klasse auch benutzen. Wir nehmen das Grundgerüst für ein Java-Programm, fügen die Klassendefinition von Mitarbeiter hinzu und erzeugen dann in der `main()`-Methode einige Instanzen unserer neuen Klassen.

Bei der Namensgebung von Methoden[1] hat es sich eingebürgert, aktive Namen (mit Verben) zu wählen und sie mit Kleinbuchstaben beginnen zu lassen. Zusätzlich lässt man sinnvolle Teilworte mit Großbuchstaben beginnen, z.B. `leseKonfigurationsDaten()`.

Instanzen

Moment mal, Instanzen? Was soll denn das sein? Denken Sie am besten an den Mitarbeiter aus der realen Welt, nachdem wir die Klasse `CMitarbeiter` modelliert haben. Die Klasse `CMitarbeiter` ist völlig abstrakt; eine Idee, eine Beschreibung, einfach nicht existent! Hugo Piepenbrink oder Erna Mustermann oder so ähnlich heißen die Menschen, die mit Ihnen zusammen in der Firma arbeiten! Sie sind die *Instanzen* des abstrakten Begriffs `CMitarbeiter`!

1. Für Konstruktoren hat man natürlich keine Wahl bei der Namensgebung, da hier der Klassenname genommen werden muss.

Anders ausgedrückt: Unsere Klasse `CMitarbeiter` stellt einen neuen Datentyp dar. Die »Werte« dieses Datentyps sind die Instanzen oder Objekte, die wir aus der Klasse erzeugen. Wie das geht, zeigt das folgende Beispiel.

Instanzen, Felder, Variablen ... die objektorientierte Terminologie kann schon recht verwirrend sein. Im Abschnitt 5.3, »Dreierlei Variablen« werden wir daher die wichtigsten Begriffe noch einmal zusammenfassen und gegenüber stellen.

Mit Klassen programmieren

Kommen wir zurück zu unserer Klasse `CMitarbeiter` und schauen wir uns an, wie wir Instanzen dieser Klasse bilden und verwenden können.

Listing 3.3: CMitarbeiter-Beispiel.java

```
class CMitarbeiter {
  String m_name;
  String m_vorname;
  int m_gehalt;

  CMitarbeiter(String name, String vorname,
               int gehalt) {
    m_name = name;
    m_vorname = vorname;
    m_gehalt = gehalt;
  }

  void datenAusgeben() {
    System.out.println("\n");
    System.out.println(" Name    : " + m_name);
    System.out.println(" Vorname : " + m_vorname);
    System.out.println(" Gehalt  : " + m_gehalt + " Euro");
  }

  void gehaltErhoehen(int erhoehung) {
    m_gehalt += erhoehung;
  }
}

public class CMitarbeiterBeispiel {
  public static void main(String[] args) {
    // 2 neue Mitarbeiter instanziieren
    CMitarbeiter billy  = new CMitarbeiter("Gates","Bill",3000);
    CMitarbeiter stevie = new CMitarbeiter("Jobs","Steve",3500);

    // Daten ausgeben
    billy.datenAusgeben();
    stevie.datenAusgeben();
```

```
    // Gehalt von a erhöhen
    billy.gehaltErhoehen(500);

    // Kontrolle
    billy.datenAusgeben();
    stevie.datenAusgeben();
  }
}
```

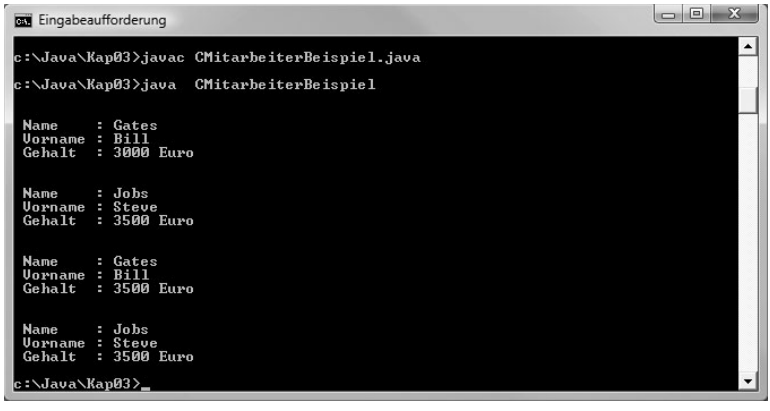

Abb. 3.2: Ausgabe des Programms CMitarbeiter-Beispiel

Sie sollten das Beispiel in den Editor eingeben, kompilieren und ausführen. Denken Sie daran, dass die Datei den gleichen Namen wie die Hauptklasse tragen muss, also in diesem Fall *CMitarbeiterBeispiel.java*. Wenn die erste Begeisterung über das funktionierende Programm vorbei ist, können Sie sich wieder setzen und die nachfolgenden Erläuterungen lesen.

Instanzen werden mit new gebildet

Das Meiste sollte Ihnen mittlerweile schon vertraut vorkommen. Spannend wird es in der `main()`-Funktion der Hauptklasse. Dort werden Instanzen der Klasse `CMitarbeiter` angelegt:

```
CMitarbeiter billy = new CMitarbeiter("Gates","Bill",3000);
```

Was macht der Compiler, wenn er diese Zeile antrifft? Nun, er legt eine neue Variable mit Namen `billy` an! Das sagt ihm die Seite links von dem Gleichheitszeichen. Die rechte Seite teilt ihm mit, dass er eine neue Instanz der Klasse `CMitarbeiter` erzeugen soll. Dazu wird mithilfe des Schlüsselwortes `new` der Konstruktor der Klasse aufgerufen, der drei Parameter erwartet, die wir ihm ordnungsgemäß übergeben.

Instanzen von Klassen müssen mit dem Operator `new` gebildet werden.

Instanzen sind Referenzen

Damit Sie später nicht den Durchblick verlieren, wollen wir an dieser Stelle etwas technischer werden, denn es besteht ein fundamentaler Unterschied zwischen den Variablenvereinbarungen

```
int billy = 4;
```

und

```
CMitarbeiter billy = new CMitarbeiter("Gates","Bill",3000);
```

int billy

Im ersten Fall wird eine `int`-Variable angelegt, d.h., der Compiler ordnet dem Namen `billy` einen bestimmten Speicherbereich zu. Gleichzeitig initialisieren wir die Variable mit dem Wert 4, wobei der Wert 4 direkt in dem Speicherbereich der Variablen abgelegt wird (siehe Abbildung 3.3).

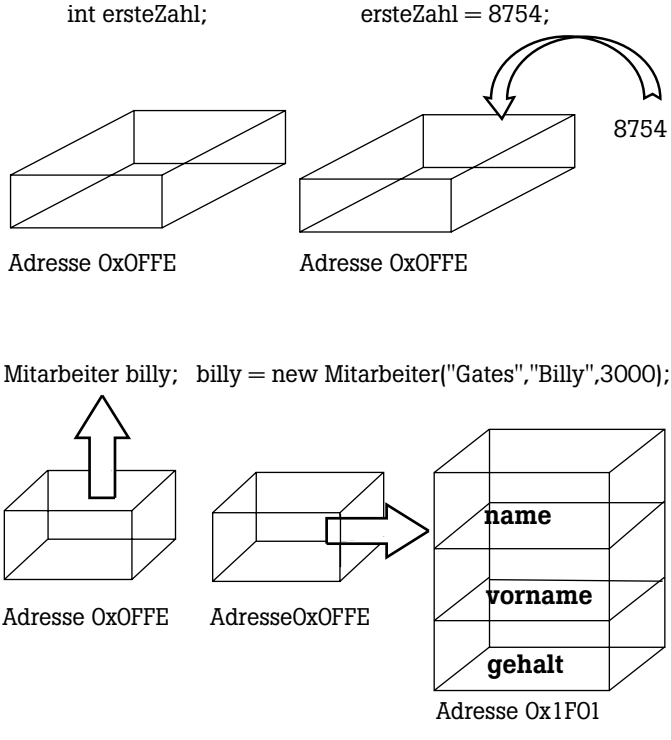

Abb. 3.3: Instanzbildung

CMitarbeiter billy

Im zweiten Fall wird zwar ebenfalls eine Variable `billy` angelegt, aber in ihr wird nicht einfach ein Wert abgelegt. Stattdessen wird mithilfe des `new`-Operators der Konstruktor der Klasse `CMitarbeiter` aufgerufen. Dieser bildet eine Instanz der Klasse, die im Speicher angelegt wird – aber nicht in dem Speicherbereich, der für die Variable `billy` eingerichtet wurde. Tat-

sächlich existiert die Instanz ganz unabhängig irgendwo im Speicher. Bei der Zuweisung der Instanz an die Variable `billy` wird dann nicht etwa der Inhalt aus der Instanz in den Speicherbereich der Variablen `billy` kopiert. Nein, stattdessen wird in der Variablen `billy` die *Adresse* des Speicherbereichs der Instanz abgespeichert. Ist dies erst einmal geschehen, sprechen wir wieder einfach von der Instanz `billy` und sehen großzügig darüber hinweg, dass `billy` eigentlich nur eine Variable ist, die eine Speicherzelle bezeichnet, in der ein Verweis (eine *Referenz*) auf die eigentliche Instanz abgespeichert ist.

Noch deutlicher werden die Vorgänge, wenn wir die Instanzbildung in zwei Schritte zerlegen:

```
CMitarbeiter billy;
billy = new CMitarbeiter("Gates","Bill",3000);
```

Zuerst wird eine Variable vom Typ `CMitarbeiter` deklariert, die zu diesem Zeitpunkt noch keinen gültigen Wert besitzt. In der zweiten Zeile wird mit dem `new`-Operator eine neue Instanz kreiert und `billy` erhält dann den Verweis auf die Speicherzellen, wo die Instanz der Klasse zu finden ist.

Dies ist eine äquivalente Möglichkeit. Meistens werden Sie in Programmen die kompakte Variante sehen. Sie wissen ja, Programmierer sind schreibfaul und lieben das Kryptische ...

Alle Variablen von Klassen sind Referenzen.

Gewöhnen Sie es sich an, Referenzen direkt mit einer Instanz zu verbinden (beispielsweise durch Aufruf des `new`-Operators oder durch Zuweisung eines Werts). Ansonsten kann es zu Fehlern kommen, wenn Sie Referenzen verwenden, die wahllos auf irgendeinen Speicherbereich und nicht auf eine konkrete Instanz verweisen.

Zugriff auf Instanzen

Wie erfolgt nun der Zugriff auf die Instanzen `billy` und `stevie`? Nehmen wir beispielsweise die Anweisung `billy.datenAusgeben()`. Man gibt einfach den Namen der Instanz an und den Namen der gewünschten Methode, verbunden durch einen besonderen Operator, den Punkt-Operator »`.`«.

Verfügt die Methode über Parametern, werden diesen in der Klammer Argumente übergeben. Wichtig bei der Parameterübergabe ist vor allem die Reihenfolge. Sie muss identisch sein mit der Reihenfolge in der Definition der Methode. Der Konstruktor der Klasse `CMitarbeiter` muss also immer zuerst zwei Zeichenketten für die Namen erhalten und dann eine Zahl für das Gehalt.

3.5 Arrays

Nun wäre es recht unpraktisch, wenn wir uns in dem Beispiel für jeden neuen Mitarbeiter, für den wir eine Instanz der Klasse `CMitarbeiter` anlegen, auch einen neuen Variablennamen ausdenken müssten. Wenn die Firma etwas größer ist, dann kommen wir schon in arge Bedrängnis. Aber glücklicherweise gibt es dafür eine Konstruktion, die man Array nennt. Am besten schauen wir uns gleich ein Beispiel für die Definition eines Arrays an:

```
int[] werte = new int[100];
```

Obige Deklaration erzeugt ein Array mit dem Namen `werte` und 100 Elementen, wobei als Elemente nur Integer-Werte erlaubt sind. Möchte man andere Werte in dem Array ablegen, tauscht man einfach den Datentyp `int` in der Deklaration gegen einen beliebigen anderen Datentyp oder eine Klasse aus:

```
int[] vektor = new int[3];
boolean[] optionen = new boolean[3400];
CMitarbeiter[] personalListe = new CMitarbeiter[4000];
double[][] matrix = new double[3][3];
```

Sicherlich ist Ihnen aufgefallen, dass der Operator `[]` bei der Array-Definition die entscheidende Rolle spielt; er gibt an, wie viele Elemente in das Array aufgenommen werden können. Mit seiner Hilfe können auch mehrdimensionale Arrays angelegt werden, die man sich als eindimensionale Arrays vorstellen kann, deren Elemente wiederum aus Arrays bestehen.

Der `[]`-Operator wird aber nicht nur bei der Definition der Arrays, sondern auch zum Zugriff auf einzelne Elemente der Arrays verwendet: man braucht lediglich in den eckigen Klammern anzugeben, auf das wievielte Element zugegriffen werden soll. Die Zahl in den Klammern nennt man daher auch Index und die Art des Zugriffs »indizierten Zugriff«.

Wenn Sie ein Array von zehn Elementen eines elementaren Datentyps deklarieren, beispielsweise `int`-Werte, dann sind in dem Array direkt zehn `int`-Werte enthalten (allerdings alle 0). Wenn Sie ein Array von zehn Objekten einer Klasse definieren, dann enthält das Array nur Null-Verweise (`null`-Referenzen). Sie müssen den einzelnen Array-Elementen erst Objekte der Klasse zuweisen.

```
vektor[0] = 4;
optionen[4] = false;
matrix[0][10] = 1.72;
```

```
// Objekte in Array ablegen
personalListe[0] = new CMitarbeiter("Schramm", "Herbert", 3500);
personalListe[1] = billy; // billy sei eine CMitarbeiter-Instanz

// Objekte in Array verwenden
personalListe[0].datenAusgeben();
personalListe[1].datenAusgeben();
```

Sie sehen, Arrays sind ganz einfach zu verwenden. Aber eine Regel müssen Sie sich besonders nachhaltig einprägen:

Das erste Element eines Arrays hat den Index 0 (in Worten NULL!). Diese seltsame Eigenschaft hat Java von der Programmiersprache C geerbt, wo sie schon Tausenden von Programmierern zahllose Stunden an Fehlersuche verursacht hat.

Wieso? Der Grund liegt wohl in der menschlichen Psyche. Wenn Sie wie oben das Array `personalListe` mit 4000 Einträgen definiert haben, erfordert es geradezu übermenschliche Kräfte, um Ihrem Gehirn die fixe Idee auszutreiben, dass der Eintrag `personalListe[4000]` existiert. Da bei der Definition des Arrays aber die Anzahl an Elementen angegeben wird und der erste Eintrag bei 0 beginnt, ist das **falsch**. Das letzte gültige Element ist `personalListe[3999]`.

Im Gegensatz zu anderen Programmiersprachen werden Sie bei Java immerhin während der Programmausführung darauf hingewiesen, dass ein Zugriff auf nicht legale Elemente eines Arrays stattfindet und der Java-Interpreter bricht ab mit der Fehlermeldung `ArrrayIndexOutOfBoundsException`.

Das erste Element eines Arrays hat immer den Index 0.

Nun sind Sie auch gerüstet, um die `main()`-Funktion, die in jedem Programmbeispiel auftaucht, etwas besser zu verstehen:

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

Wie Sie mittlerweile wissen, stehen in den runden Klammern die Parameter, die diese Funktion erwartet. `String[] args` bedeutet, dass `main()` ein Array von String-Objekten als Parameter erwartet. In Kapitel 6.6 werden wir noch ein kleines Beispiel dazu sehen, wie man mithilfe von `args` Kommandozeilenargumente einliest und innerhalb des Programms verarbeitet.

Arrays sind Klasseninstanzen

Noch eine letzte Bemerkung zu Arrays: Jedes Array, das Sie anlegen, ist selbst automatisch eine Instanz der Klasse `Array` (auch so eine von den vie-

len schon mitgelieferten Klassen in Java, allerdings eine ganz besondere). Es gibt daher auch Methoden und Felder, auf die Sie zugreifen können (wenn man sie kennt!). Ein sehr nützliches Feld ist beispielsweise `length`, es liefert die Größe des Arrays zurück:

```
CMitarbeiter[] personalListe = new CMitarbeiter[100];
int anzahlElemente;
// ....

anzahlElemente = personalListe.length;

// Gibt die Größe aus, also 100
System.out.println("Array Größe ist " + anzahlElemente);
```

Nach diesem ersten intensiven Kontakt mit Klassen wenden wir uns in dem nächsten Kapitel wieder anderen Grundbestandteilen von Java zu (obwohl Klassen uns auch da begegnen werden), bevor Sie in Kapitel 5 in die Tiefen der objektorientierten Programmierung eintauchen!

3.6 Vordefinierte Klassen und Pakete

Zum Schluss aber noch einige wichtige Informationen über die Klassen, die schon fix und fertig in Java integriert sind wie die `String`-Klasse. Diese Klassen sind in logische Gruppen sortiert, die sich *Pakete* nennen. Im neuen Java 6 Standard (den dieses Buch behandelt) gibt es über Dutzende von Paketen mit weit über 1000 Klassen! Eine immense Zahl, nicht wahr? Alle werden wir im Laufe des Buches nicht kennenlernen, aber die wichtigsten und nützlichsten. Danach werden Sie als alter Java-Hase kein Problem mehr haben, in der Java-API-Dokumentation herumzustöbern und hilfreiche Klassen zu entdecken und in Ihre Programme einzubauen, evtl. sogar zu modifizieren. Ja, auch das geht (meistens jedenfalls)!

Was muss man tun, um solche fertigen Klassen in eigenen Programmen zu verwenden? Ganz einfach: Entweder man stellt überall im Quellcode dem Klassennamen den Paketpfad voran oder man *importiert* den Klassennamen und verwendet die Klasse danach einfach so, als hätte man Sie selbst definiert

Die `String`-Klasse befindet sich beispielsweise im Paket *java.lang*. In unser Programm importieren wir sie mithilfe der Anweisung:

```
import java.lang.String;
```

Meistens braucht man mehrere Klassen aus einem Paket. Anstatt nun jede einzelne Klasse explizit zu importieren, kann man auch alle Klassennamen aus dem Paket mithilfe des *-Symbols importieren:

```
import java.lang.*;
```

Bestimmt nagen im Moment an Ihnen die Zweifel, ob Sie das richtige Buch lesen. Wieso hat denn das Beispiel von vorhin geklappt? Da war weit und breit kein Import-Schnickschnack. Es geht wohl doch ohne! Was also soll das alles? Nun ja, es gibt ein Paket, das automatisch vom Compiler importiert wird, weil in ihm so viele wichtige und immer wieder benötigte Klassen sind, dass praktisch kein Programm ohne es auskäme. Und jetzt raten Sie mal, wie dieses Paket heißt! Genau. Aber für die anderen Pakete gilt das oben Gesagte. Sie müssen die Klassennamen explizit importieren oder sich die Mühe machen, die Klassennamen mit Paketpfad zu schreiben

Eine Frage der Eindeutigkeit

Welcher Sinn liegt darin, die Klasse in Pakete aufzuteilen? Stellen Sie sich vor, Sie arbeiten mit anderen Programmierern zusammen an einem größeren Objekt. Jeder bearbeitet einen kleinen Teilaspekt der Anwendung und definiert dafür seine eigene Klassen, die später zur Gesamtanwendung zusammengefasst werden. Gut möglich, dass dabei zwei Programmierer Klassen gleichen Namens, beispielsweise `Vektor`, definiert haben. Wie aber soll der Compiler beim Übersetzen des Gesamtprogramms dann wissen, welche `Vektor`-Klasse wann gemeint ist.

Sind die beiden `Vektor`-Klassen in unterschiedlichen Paketen definiert, können sie dagegen problemlos beide verwendet und durch die Paketpfade voneinander unterschieden werden:

```
projekt.jim.Vektor v = new projekt.jim.Vektor();   // Vektor-Klasse
                                                   // des einen
                                                   // Programmierers
projekt.kurt.Vektor v = new projekt.kurt.Vektor(); // Vektor-Klasse
                                                   // des anderen
                                                   // Programmierers
```

Oder es wird eine `Vektor`-Klasse importiert und die andere bei Bedarf über den Paketpfad referenziert:

```
import projekt.jim.Vektor;
...
Vektor v = Vektor();
projekt.kurt.Vektor v = new projekt.kurt.Vektor();
```

Das Einzige, was man nicht tun darf, ist beide Klassennamen in ein gemeinsames Paket zu importieren (und dadurch die eindeutige Zuordnung wieder aufzuheben).

3.7 Zusammenfassung

Daten werden in Programmen durch Variablen repräsentiert. Jeder Variablen entspricht ein Speicherbereich, in dem der aktuelle Wert der Variablen abgelegt wird. Variablen müssen deklariert werden, um dem Compiler den Namen der Variablen bekannt zu machen. Bei der Deklaration der Variablen wird auch der Datentyp der Variablen angegeben, der festlegt, welche Werte die Variable aufnehmen kann und welche Operationen auf der Variablen durchgeführt werden können.

Klassen definieren Eigenschaften (Felder) und Verhaltensweisen (Methoden). Instanzen von Klassen werden mit dem Operator `new` und durch Aufruf des Konstruktors der Klasse gebildet.

Java-Programme sind Ansammlungen von Klassendefinitionen.

3.8 Fragen und Antworten

1. **Datentypen sind das A und O der Variablendeklaration. Zählen Sie die Schlüsselwörter auf, mit denen die verschiedenen Datentypen bei der Variablendeklaration spezifiziert werden.**

   `int`, `short`, `byte`, `long`, `char`, `boolean`, `float`, `double`

2. **Welche der folgenden Variablennamen sind nicht zulässig?**

   ```
   123
   zähler
   JW_Goethe
   JR.Ewing
   _intern
   double
   Liebe ist
   ```

   Die folgenden Bezeichner sind nicht zulässig:

   ```
   123         // Ziffer als erstes Zeichen
   JR.Ewing    // Punkt in Namen
   double      // reserviertes Schlüsselwort
   Liebe ist   // Leerzeichen in Namen
   ```

3. **Welche der folgenden Variablendeklarationen sind nicht zulässig?**

   ```
   int 123;
   char c;
   boolean option1, option2;
   boolean option1 option2;
   ```

```
short y = 5;
short x = 5+1;
short x = y; // y wie oben
```

Die folgenden Deklarationen sind nicht zulässig:

```
int 123;                 // ungültiger Name
boolean option1 option2; // fehlendes Komma
short x = y;             // x zuvor deklariert
```

4. Warum führt der folgende Code zu einem Compiler-Fehler?

```
long x;
x = 5;
long x;
x = 4;
```

Die Variable `x` wird zweimal definiert. Würde der Compiler diesen Anweisungen folgen, würde er für jede Definition einen Speicherbereich reservieren und mit dem Namen `x` verbinden. Die Variable `x` wäre dann mit zwei Speicherbereichen verbunden. Dies darf aber nicht sein – zwischen einer Variablen und ihrem Speicherbereich muss immer eine eindeutige Beziehung bestehen.

5. Die Division haben Sie in Abschnitt 3.3 kennengelernt. Erinnern Sie sich noch an den Unterschied zwischen `250/4` und `250.0/4`?

250/4 ist eine Division von Ganzzahlen. Der Compiler liefert daher auch ein ganzzahliges Ergebnis zurück: 62. Im Falle von 250.0/4 wandelt der Compiler alle Werte in `double`-Fließkommazahlen um. Das Ergebnis lautet daher 62.5.

6. Warum rechnet der Computer mit Binärzahlen?

Computer sind elektronische Rechner, in denen Daten in Form von Spannungswerten verarbeitet werden. Einfacher als die Unterscheidung verschiedener Spannungswerte ist die Entscheidung, ob überhaupt Spannung vorhanden ist oder nicht. Ja oder Nein, Null oder Eins. Darum werden alle Daten binär codiert.

7. Was sind Klassen?

Klassen sind spezielle Datentypen, in denen verschiedene Variablen und Methoden zusammen deklariert werden können. In Java bestehen Programme praktisch nur aus der Definition von Klassen.

8. Welche Beziehung besteht zwischen einer Klasse und ihren Instanzen?

Klassen sind Datentypen, Instanzen sind »Werte« von Klassen.

9. Welches Paket muss nicht explizit importiert werden?

Das Paket `java.lang` braucht nicht explizit importiert zu werden.

3.9 Übungen

1. Angenommen, Sie wollten einen Flugsimulator schreiben. Ihre Szenerie ist ganz einfach aufgebaut und besteht praktisch nur aus einem Untergrund und drei Wassertürmen, die umflogen werden sollen. Überlegen Sie sich, welche Klassen Sie für diesen Flugsimulator definieren müssen, welche Felder und Methoden benötigt werden und welche Instanzen Sie erzeugen müssen.
2. Angenommen, Sie haben von einem anderen Programmierer eine Klasse `CAuto` mit den Eigenschaften `m_geschwindigkeit` und `m_benzinverbrauch` sowie den Methoden `anlassen()`, `beschleunigen()` und `bremsen()`. Sie wissen sonst nichts über die Implementierung der Klasse. Wie rufen Sie die Methode `anlassen()` auf?
3. Was passiert in obiger Aufgabe, wenn Sie die Methode `bremsen()` aufrufen, bevor Sie die Methoden `anlassen()` und `beschleunigen()` aufgerufen haben?