Kapitel 3

Java-Grundkurs: Datenverarbeitung

Im vorangehenden Kapitel haben wir bereits ein erstes Programm aufgesetzt, ein Programm mit grafischer Benutzeroberfläche, mit Label, Schalter und Eingabefeld. Für all dies mussten wir nicht einmal richtig programmieren – dank der RAD-Umgebung des JBuilder konnten wir uns alles mit der Maus zusammenklicken. Als wir aber daran gingen, eine Ereignisbehandlung für das Drücken des Schalters aufzusetzen, haben wir die Grenzen der RAD-Umgebung kennen gelernt. Sie ist wunderbar geeignet, um grafische Benutzeroberflächen zusammenzustellen oder Software-Bausteine, die als Komponenten (oder JavaBeans) zur Verfügung stehen, in ein Programm zu integrieren, doch wenn es darum geht, den funktionellen Code aufzusetzen, der hinter der schönen grafischen Oberfläche steht, dann sind wir wieder auf uns allein gestellt. Dann hilft uns keine RAD-Umgebung mehr, dann helfen nur noch gute Java-Kenntnisse.

Die nächsten drei Kapitel sind daher ganz der grundlegenden Java-Syntax gewidmet.

Sie lernen in diesem Kapitel,

- ✘ was Konsolenanwendungen sind und wie man sie erstellt
- ✘ wie ein Java-Programm mit Daten umgeht,
- ✘ was Variablen, Konstanten und Anweisungen sind,
- ✘ was Datentypen und Operatoren sind,
- ✘ was Klassen sind,
- ✘ was Arrays sind,
- ✘ welche Bedeutung den Packages zukommt

3.1 Konsolen-Anwendungen

Man kann bei einem Java-Programm zwei Grundformen unterscheiden: es gibt *Applikationen* und *Applets*.

- ✘ Eine Applikation oder Anwendung ist ein vollwertiges und vor allem eigenständiges Programm, das Sie auf Ihrem Rechner mit Hilfe des Java-Interpreters *java.exe* oder auch innerhalb der JBuilder-Umgebung[1] ausführen können.
- ✘ Ein Applet[2] ist im Gegensatz zur Applikation unselbstständig, da es nicht direkt mit dem normalen Java-Interpreter ausgeführt werden kann. Das Applet benötigt einen ganz bestimmten »Lebensraum«, eine Umgebung, in der es ablaufen kann. Diese Umgebung stellt in der Regel ein WWW-Browser (beispielsweise Netscape oder Internet Explorer) zur Verfügung. Daher kennen Sie Applets sicher vom Surfen im Internet, zum Beispiel werden beim Homebanking noch häufig Java-Applets eingesetzt. Applets sind hinsichtlich ihrer Programmierung nur ein Spezialfall von normalen Java-Applikationen. Wir werden sie in Kapitel 14 zur Applet-Programmierung näher betrachten.

Die Applikationen kann man weiter unterscheiden, je nachdem ob sie über eine grafische Benutzeroberfläche (also Fenster, Schalter, etc.) verfügen oder ohne eine solche auskommen. Java-Programme der letzteren Art nennt man auch *Konsolenprogramme*, weil Ein- und Ausgaben nur über eine Konsole[3] erfolgen können.

Reine Konsolenprogramme werden heutzutage eigentlich nur noch für Aufgaben verwendet, die unsichtbar im Hintergrund erledigt werden können – beispielsweise für die Implementierung von Webservern. In der bunten, multimedial-verwöhnten Welt der Endanwender spielen Konsolenprogramme keine große Rolle mehr, da heute jedermann nach Programmen verlangt, die grafische Benutzeroberflächen haben und unter Windowing-Systemen wie Microsoft Windows oder XWindows (bzw. Gnome, KDE, etc.) laufen. Trotzdem werden wir uns in diesem und dem nächsten Kapitel ausschließlich mit Konsolenprogrammen beschäftigen, und zwar aus didaktischen Gründen. Wenn wir uns in der Folge mit der Deklaration von Variablen, mit Operatoren, Bedingungen und Schleifen oder mit den Grundkonzepten der objektorientierten Programmierung beschäftigen, soll uns nichts von der ei-

---

1 Der JBuilder ruft dann für Sie java.exe auf.

2 Ein Kunstwort, das soviel wie »kleine Applikation« bedeutet.

3 Unter Windows ist dies das MS-DOS Fenster mit dem schwarzen Hintergrund.

gentlichen Thematik ablenken. Darum verzichten wir auf die grafische Oberfläche und konzentrieren uns ganz auf den funktionellen Code.

Das Grundgerüst

Das minimale Grundgerüst einer Java-Anwendung ohne grafische Benutzeroberfläche ist erstaunlich kurz. Tatsächlich braucht man im Grunde ja nichts anderes als eine `main()`-Methode. Da die Programmausführung, wie Sie aus Kapitel 2 bereits wissen, mit dem Aufruf dieser Methode beginnt, braucht man nur den gesamten Programmcode in den Rumpf dieser Methode zu schreiben.

Listing 3.1: Ein minimales Anwendungsgerüst

```
class CHalloWelt
  {
  public static void main(String[] args)
    {
    System.out.println("Hallo Welt!");
    }
  }
```

Listing 3.1 zeigt uns wie das kleinstmögliche ausführbare Java-Programm aussieht. es besteht aus einer einzigen Klasse, die als einziges Klassenelement die statische Methode `main()` enthält.

Wenn das Java-Programm gestartet wird, beginnt der Interpreter die Ausführung, indem er die `main()`-Methode sucht und dort zur ersten Anweisung (in unserem Beispiel also zu `System.out.println()`) springt und diese ausführt. Danach werden schrittweise alle weiteren Anweisungen ausgeführt, bis die schließende Klammer der `main()`-Methode erreicht wird. Im obigen Miniprogramm gibt es nur eine Anweisung, die den String »`Hallo Welt`« ausgibt.

In einem Java-Programm darf es nur eine einzige Klasse mit einer statischen `main()`-Methode geben.

Konsolenanwendungen im JBuilder

Nun wird es Zeit, dieses Programm mit dem JBuilder zu erstellen. Leider bietet der JBuilder derzeit keine direkte Unterstützung für Konsolenanwendungen, das heißt es gibt in der OBJEKTGALERIE (Aufruf über DATEI/NEU) kein Symbol KONSOLENANWENDUNG. Das ist aber kein Beinbruch. Wir können uns auf verschiedenen Wegen behelfen.

- Die erste Möglichkeit bestünde darin, ganz auf den JBuilder zu verzichten und stattdessen auf die JDK-Tools zurückzugreifen. Den Quelltext würden Sie dann in einem einfachen Texteditor aufsetzen und unter dem Namen der Klasse und mit der Extension .java abspeichern – Den Quelltext würden Sie also in der Datei *CHalloWelt.java* abspeichern. (Achtung! Der Name der Klasse und der Name der Datei müssen identisch sein, wobei auch die Groß- und Kleinschreibung zu beachten ist!)

  Danach öffnen Sie ein Konsolenfenster (unter Windows ist dies die MSDOS-Eingabeaufforderung, die über START/PROGRAMME aufgerufen werden kann) und wechseln in das Verzeichnis der Java-Quelltextdatei. Unter der Voraussetzung, dass das Verzeichnis, in dem die JDK-Tools stehen, im Pfad Ihres Systems eingetragen ist, können Sie dann nacheinander den Java-Compiler und den Java-Interpreter aufrufen:

  ```
  Prompt> javac CHalloWelt.java

  Prompt> java CHalloWelt
  ```

- Die zweite Möglichkeit bestünde darin, wie in Kapitel 2 ein Projekt mit dem JBuilder-Anwendungsgerüst anzulegen. Dabei müssen Sie im zweiten Dialog des Anwendungsexperten darauf achten, dass kein Package eingerichtet wird (einfach den Eintrag im Eingabefeld löschen) und dass Sie für die Anwendungsklasse den Namen der Klasse aus dem Beispiellisting (in obigem Fall also `CHalloWelt`) angeben, damit der JBuilder eine gleichlautende .java-Quelldatei anlegt. Dann brauchen Sie nur noch den Inhalt der Datei mit der Anwendungsklasse (in unserem Fall *CHalloWelt.java*) durch den Inhalt des Beispiellistings ersetzen. Die Datei *Frame1.java* können Sie aus dem Projekt löschen (über den entsprechenden Befehl im Kontextmenü des Projektknotens).

- Die dritte Möglichkeit ist unserer Einschätzung nach die Beste. Wir fertigen uns ein eigenes Projekt nach Maß:

1. Legen Sie mittels DATEI/NEUES PROJEKT ein neues Projekt namens *HalloWelt* an.

   Geben Sie wie bereits in Abschnitt 2.1 beschrieben auf der ersten Seite des Projekt-Experten Name, Typ, Pfad und Verzeichnis des Projekt an.

   Klicken Sie danach auf FERTIG STELLEN.

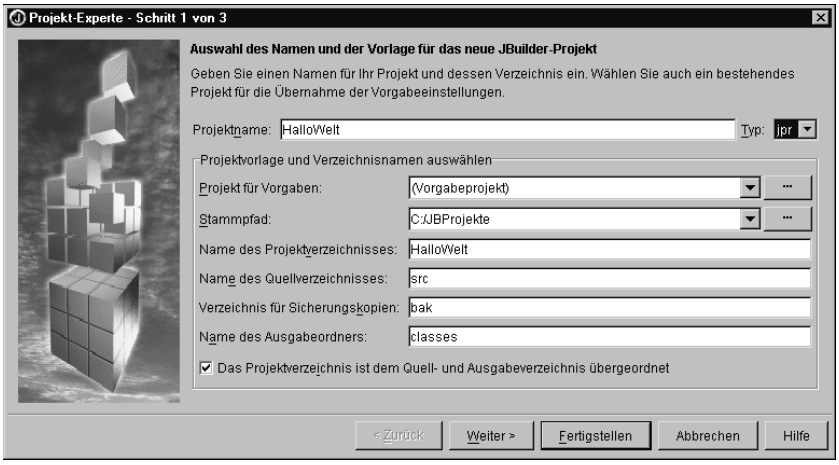


Abb. 3.1: Neues leeres Projekt anlegen

2. Erweitern Sie das Projekt um eine Quellcodedatei. Rufen Sie dazu wieder den Befehl DATEI/NEU auf und wählen Sie in der Objektgalerie den Eintrag KLASSE. Es erscheint der Klassen-Experte.

   Löschen Sie den Eintrag im Feld PACKAGE (für unsere einfachen Programmierbeispiele ist es nicht notwendig, die Klassen auf eigene Packages zu verteilen.)

   Geben Sie den Namen der einzurichtenden Klasse ein.

   Aktivieren Sie die Option MAIN-FUNKTION GENERIEREN.

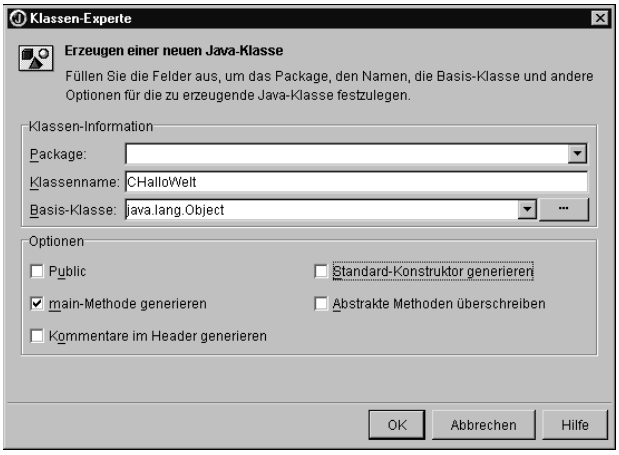


Abb. 3.2: Neue Datei mit Klassengerüst anlegen

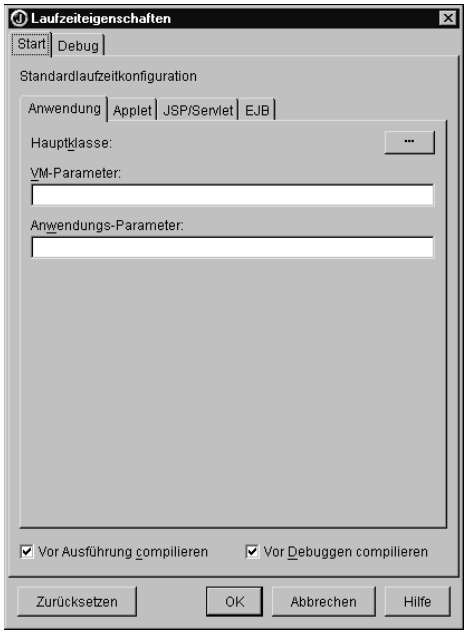
3. Jetzt können Sie das vom JBuilder angelegte Klassengerüst entsprechend des jeweiligen Beispiellistings ausbauen.

Wenn Sie die Listings nicht nachprogrammieren wollen (was wir Ihnen allerdings empfehlen), werfen Sie einmal einen Blick auf Ihre Buch-CD. Dort finden Sie eine Zusammenstellung der wichtigsten Beispielprogramme.

4. Nun können Sie das Programm vom JBuilder ausführen lassen. Rufen Sie dazu den Befehl START/PROJEKT AUSFÜHREN auf.

Wenn Sie das Programm das erste Mal ausführen, fragt Sie der JBuilder nach der HAUPTKLASSE.

Abb. 3.3: Hauptklasse angeben



Der Grund hierfür ist der, dass in einer Java-Datei mehrere Klassen gleichzeitig definiert sein können und der JBuilder wissen muss, welche Klasse er dem Java-Interpreter zum Starten des Programms übergeben muss.

5. Helfen Sie dem JBuilder also auf die Sprünge, klicken Sie auf die Schaltfläche mit der Ellipse (...) und wählen Sie aus der erscheinenden Liste die Klasse aus Ihrem Beispielcode aus.

Danach wird das Programm ausgeführt. Im Meldungsfenster erscheint die Ausgabe der `System.out.println()`-Anweisung:

```
Hallo Welt!
```

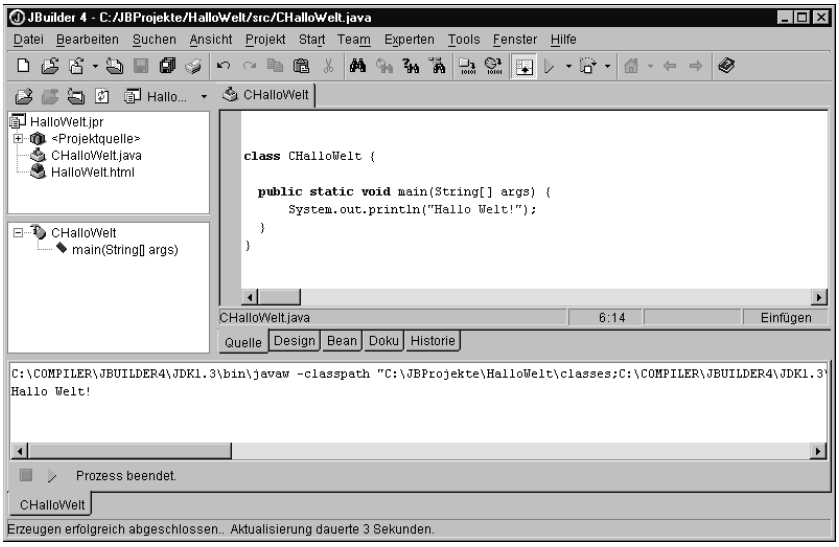


Abb. 3.4: Ausführung des Beispielprogamms im JBuilder

Wenn Sie die nachfolgenden Beispiele nachprogrammieren wollen, verfahren Sie analog.

3.2 Datentypen und Variablen

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.

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. 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) – konnte nur stattfinden, weil man einen Ausweg fand. Dieser bestand einfach darin, es der Software – dem laufenden Programm – zu überlassen, die vom Anwender eingegebenen Daten (seien es Zahlen, Text, Bilder etc.) ins Binärformat umzuwandeln und umgekehrt die auszugebenden Daten wieder vom Binärformat in eine leicht lesbare Form zu verwandeln.

»Gemeinheit«, werden 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 static void main(String[] args)
    {
    int erste_Zahl;
    int zweite_Zahl;
    int ergebnis;

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

Obiger Quellcodeauszug stellt kein vollständiges Programm dar – es fehlt die umliegende Klasse, in der die `main`-Methode definiert sein muss (siehe Listing 3.1).

Wenden wir unsere Aufmerksamkeit den Vorgängen in der `main()`-Methode zu. Dort werden zuerst die für die Berechnung benötigten Variablen deklariert.

Variablen

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.

```
int erste_Zahl;
```

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

```
erste_Zahl = 8754;
```

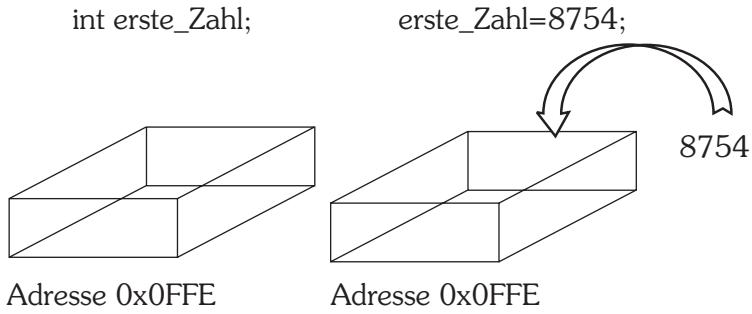


Abb. 3.5: Deklaration und Zuweisung

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 `erste_Zahl` nach der Anweisung

```
erste_Zahl = 8754;
```

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

```
erste_Zahl = 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 mit Hilfe 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`-Variable `erste_Zahl` gleich 5 ist, wir sagen einfach, `erste_Zahl` ist gleich 5. Wir sprechen nicht davon, dass wir mit Hilfe 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 Wertes 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*, das heißt, ihre Werte werden in Anweisungen ausgelesen oder neu gesetzt.

Listing 3.2: CVariablen.java

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

    ersteZahl = 8754;
    zweiteZahl = 398;
    ergebnis = ersteZahl - zweiteZahl;
    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, erfolgte aber erst in einem zweiten Schritt, der so genannten 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 »Methoden von Klassen«, und »Viererlei Variablen«, wo wir diese Frage klären werden. Im Moment, da wir uns nur mit so genannten 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, das heißt 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
	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	-3,40282347E+38 bis +3,40282347E+38
double[4]	Fließkommazahl	-1,7976931348623157E+308 bis +1,7976931348623157E+308

4 Für Fließkommazahlen (Zahlen mit Nachkommaanteil) sollten Sie generell den Datentyp `double` verwenden. Mit diesem lassen sich nicht nur größere Zahlen darstellen, kleinere Zahlen lassen sich zudem in größerer Präzision (mit mehr Nachkommastellen) darstellen. Falls Geschwindigkeit allerdings eine Rolle spielt, sind `float`-Werte schneller!

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 65.536 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 zum Beispiel 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 krumme_Zahl;
boolean ja, nein, oder_doch;
boolean Antwort;
short klein = -4;
char buchstabe;
char Ziffer;

ganzeZahl = 3444;
krumme_Zahl = 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, das heißt 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 Beamte

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 Compiler verderben wollen, sollten Sie insbesondere auf folgende Punkte achten:

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

```
krumme_Zahl = 47.11;
```

- Java unterscheidet streng zwischen *Groß- und Kleinschreibung*. Wenn Sie also eine Variable namens `krumme_Zahl` deklariert haben, dann müssen Sie auch `krumme_Zahl` schreiben, wenn Sie auf die Variable zugreifen wollen, und nicht `krumme_zahl`, `Krumme_Zahl` oder `KRUMME_ZAHL`.

Und natürlich gibt es auch Regeln für die Einführung von Bezeichnern, also beispielsweise Variablennamen. Wir wollen uns an dieser Stelle aber nicht mit den endlosen Vorschriften befassen, welche die Konstruktion von Variablennamen regeln. Merken Sie sich einfach Folgendes:

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

Sie dürfen also Ihre Variable nicht `class` nennen, da dies ein reserviertes Wort ist. Im Anhang finden Sie eine Liste mit den verbotenen Begriffen. Neben Buchstaben und Ziffern sind fast alle Unicode-Zeichen erlaubt. Insbesondere die Umlaute sind erlaubt. 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 gewissen Plattformen zu Schwierigkeiten bei der Kompilation führen kann.

3.3 Variablen versus Konstanten

Muss man wirklich erst erwähnen, dass man den Wert einer Variablen ändern kann, indem man ihr einen neuen Wert zuweist (das heißt einen neuen Wert in ihren Speicherbereich schreibt), während der Wert einer Konstanten 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 so genannten *Literalen*.

```
krumme_Zahl = 47.11; // Zuweisung eines Literals
krumme_Zahl = 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',
`char` (Sonderzeichen)	'\n', '\\',
`char` (Unicode)	'\u1234'
`String`	"Dies ist ein String"
`int`	12, -128
	oktal: 077
	hexadezimal: 0xFF1F
`long`	12L, 1400000
`float`	12.4f, 10e-2f
`double`	47.11, 1e5

Zweitens:

Sie deklarieren eine Variable mit dem Schlüsselwort `final`:

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

Da der Wert einer solchen »konstanten Variablen« nach der Deklaration nicht mehr verändert werden kann, folgt zwangsläufig, dass sie bei der Deklaration initialisiert werden muss.

3.4 Operatoren

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

Listing 3.3: COperatoren

```
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 genauso, 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, das heißt, die Variable `x` enthält nach Ausführung dieser Zeile als neuen Wert die Summe aus ihrem alten Wert und `z`.

Da das Hinzuaddieren eines Wertes 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 zum Beispiel 1
– +	Subtraktion, Addition	Subtraktion und Addition
<= < > >=	Vergleich	Zum Vergleich zweier Werte. Die Operatoren liefern true oder false zurück.

Tabelle 3.3: Operatoren (Fortsetzung)

Operator	Beschreibung	Beispiel
== !=	Vergleich (gleich ungleich)	Zum Vergleich auf Gleichheit oder Ungleichheit. Die Operatoren liefern true oder false zurück.
&&	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`

~	bitweises Komplement	Umkehrung der Binärpräsentation einer Zahl.
		`var1 = 1;  // ...0001`
		`var2 = ~var1;  // ...1110`

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«.

Ausdrücke

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));
```

Falls Sie mehr Mathematik benötigen als die bisher genannten Operatoren bieten, dann seien Sie getröstet. Es gibt noch eine Reihe von mathematischen Funktionen für alle gängigen Probleme wie Sinus, Kosinus, Quadratwurzel usw. Hier eine Aufstellung der wichtigsten:

Tabelle 3.4: Mathematische Funktionen

Funktion	Berechnung
double Math.acos(double x)	Arkuskosinus von x
double Math.asin(double x)	Arkussinus von x
double Math.atan(double x)	Arkustangens von x
double Math.cos(double x)	Kosinus von x
double Math.exp(double x)	e hoch x
double Math.log(double x)	Logarithmus zur Basis e von x
double Math.pow(double x, double y)	x hoch y
double Math.sin(double x)	Sinus von x
double Math.sqrt(double x)	Quadratwurzel aus x
double Math.tan(double x)	Tangens von x

Die Angabe von beispielweise `double Math.tan(double x)` bedeutet dabei, dass als Argument ein `double`-Wert erwartet wird und als Ergebnis wird ebenfalls ein `double`-Wert zurückgeliefert.

Der Einsatz erfolgt genau wie Sie es von der Schule her noch kennen, zum Beispiel

```
double y;
double z = 100;
y = Math.sqrt(z);
```

3.5 Typumwandlung

Damit wissen Sie schon fast alles, was ein guter Java-Programmierer über Variablen, Operatoren und einfache Datentypen (nennt man manchmal auch elementare 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-Wertes 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.

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

Explizite Typumwandlung

Um eine Typumwandlung zu erzwingen, die der Compiler nicht automatisch unterstützt, stellt man einfach dem zu konvertierenden Wert den gewünsch-

ten 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 zu Stande?

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 Wertes 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 Beamtenmentalität des Compilers. Er wertet Schritt für Schritt und streng nach Vorschrift die Ausdrücke 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, das heißt, 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, führt allerdings der Datentyp `float` zur schnelleren Abarbeitung.

3.6 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 so genannten Funktionen zu bündeln und auszulagern. 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 Objektes, und die Funktionen (die man dann per Konvention als Methoden bezeichnet) 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`-Variable, vor. Die Methoden, 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 zum Teil 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
  {
  }
```

Sie haben es bestimmt schon bemerkt. Wir lassen alle Klassen, die wir selbst anlegen, mit einem `C` beginnen. Das erleichtert manchmal die Orientierung in größeren Programmen. Man kann dann direkt erkennen, ob die Klassen von uns oder vom JBuilder angelegt worden sind.

Nun müssen wir aber 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;       // Monatsgehalt
  }
```

Langsam nimmt unsere Klasse konkrete Formen an! Den Datentyp `int` kennen Sie ja schon. `String` ist kein einfacher Datentyp (daher haben wir ihn im vorigen Abschnitt auch nicht kennen gelernt), sondern ebenfalls eine Klasse, genau wie `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.

Instanzvariablen

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

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;       // Monatsgehalt

  CMitarbeiter(String pName, String pVorname,
               int Gehalt)
    {
    m_name = pName;
    m_vorname = pVorname;
    m_gehalt = pGehalt;
    }

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

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

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

Bevor wir uns diese drei Methoden im Einzelnen anschauen wollen, sollten wir uns überlegen, wie eine Methodendeklaration im Allgemeinen aussehen sollte. 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:

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;
    }
```

Bezeichner

Mittlerweile haben wir die dritte Art von *Bezeichnern* (Namen, die der Programmierer einführt und per Deklaration dem Compiler bekannt gibt) kennen gelernt. 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 *Instanzvariablen* 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, so genannte *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 so genannte *Parameter* vor. Dies sind Variablen, die innerhalb der Klammern der Methodendeklaration deklariert werden. Bei Aufruf der Methode werden diesen Parametern Werte übergeben (die so genannten 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 mit Hilfe 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 + " DM");
   }
```

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 mit Hilfe 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 pErhoehung)
    {
    m_gehalt += pErhoehung;
    }
```

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 `Erhöhung`, 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 pName, String pVorname,
             int pGehalt)
    {
    m_name = pName;
    m_vorname = pVorname;
    m_gehalt = pGehalt;
    }
```

Dies ist eine ganz besondere Methode, nämlich ein *Konstruktor*. Jede Klasse braucht einen oder sogar mehrere Konstruktoren, die beim Initialisieren der Variablen (Objekte) 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 zugehörigen Instanzvariablen. Ansonsten könnte der Compiler die Zuweisungen nicht ordnungsgemäß ausführen.

Jede Klasse braucht zumindest einen Konstruktor zur Initialisierung ihrer Instanzvariablen. 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.

Instanzen und Objekte

Moment mal, Instanzen? Objekte? Was soll denn das sein? Denken Sie am besten an den Mitarbeiter aus der realen Welt, nach dem 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*, die realen Manifestationen des abstrakten Begriffs `CMitarbeiter`!

Anders ausgedrückt: Unsere Klasse `CMitarbeiter` ist keine Variable, sondern stellt einen neuen Datentyp dar. Der Wertebereich dieser Klasse sind nicht irgendwelche Zahlen (wie für `int`) oder Zeichen (wie für `char`), sondern die Instanzen, die wir mit Hilfe des Konstruktors der Klasse erzeugen.

Und was sind Objekte? »Objekt« ist einfach ein synonymer Begriff für »Instanz«, der in der objektorientierten Terminologie ebenfalls weit verbreitet ist. Manche Autoren sprechen lieber von Objekten, andere ziehen den Begriff Instanz vor. Wir halten es in diesem Buch so, dass wir von Instanzen sprechen, wenn es um programmtechnische Dinge geht (Instantiierung von Klassen, etc.), und von Objekten reden, wenn wir allgemeine Konzepte der objektorientierten Programmierung beschreiben.

Mit Klassen programmieren

Kommen wir zurück zu unserer Klasse `CMitarbeiter` und schauen wir uns an, wie wir Instanzen dieser Klasse bilden und verwenden können. Zur Vereinfachung haben wir hier `CMitarbeiter` in der gleichen Datei definiert wie die Hauptklasse `CMitarbeiterBeispiel`:

Listing 3.4: CMitarbeiterBeispiel

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

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

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

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

class CMitarbeiter
  {
  String m_name;
  String m_vorname;
  int m_gehalt;       // Monatsgehalt

  CMitarbeiter(String pName, String pVorname,
            int pGehalt)
     {
     m_name = pName;
     m_vorname = pVorname;
     m_gehalt = pGehalt;
     }

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

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

Sie sollten mal ein neues Projekt anlegen und die Hauptklasse entsprechend umbauen. 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()`-Methode 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 mit Hilfe 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.

Variablen von Klassentypen 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

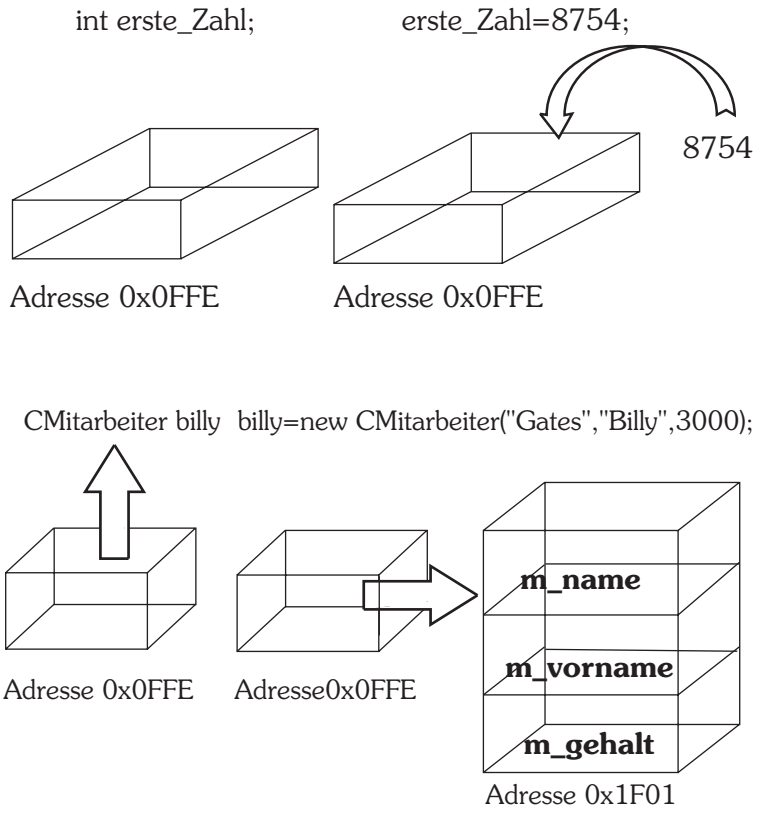
Im ersten Fall wird eine `int`-Variable angelegt, das heißt, der Compiler ordnet dem Namen `billy` einen bestimmten Speicherbereich zu. Gleichzeitig initialisieren wir die Variable mit dem Wert 4, wobei der Wert 4 direkt in den Speicherbereich der Variablen abgelegt wird (siehe Abbildung 3.6).

CMitarbeiter billy

Im zweiten Fall wird zwar ebenfalls eine Variable `billy` angelegt, aber in ihr wird nicht einfach ein Wert abgelegt. Stattdessen wird mit Hilfe 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. Tatsächlich 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 Variable `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.

Abb. 3.6: Instanzbildung



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 Wertes). 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 Parameter 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.

3.7 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 Feld oder Array nennt. Am besten schauen wir uns gleich ein Beispiel für die Definition eines Arrays an:

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

Obige Deklaration erzeugt ein Array mit dem Namen `feld` 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[] feld_3 = 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 10 Elementen eines elementaren Datentyps deklarieren, beispielsweise `int`-Werte, dann sind in dem Array direkt 10 `int`-Werte enthalten (allerdings alle 0). Wenn Sie ein Array von 10 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;
feld_3[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 Array 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 Feldes 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()`-Methode, 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 6.7 werden wir noch ein kleines Beispiel dazu sehen, wie man mit Hilfe von `args` Kommandozeilenargumente einliest und innerhalb des Programms verarbeitet.

Arrays sind Klasseninstanzen

Noch eine letzte Bemerkung zu den Arrays: Jedes Array, das Sie anlegen, ist selbst automatisch eine Instanz der Klasse `Array` (auch so eine von den vielen schon mitgelieferten Klassen in Java, allerdings eine ganz besondere). Es gibt daher auch Methoden und Variablen, auf die Sie zugreifen können (wenn man sie kennt!). Eine nützliche Instanzvariable heißt `length` und liefert die Größe des Feldes 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 wir in Kapitel 5 in die Tiefen der objektorientierten Programmierung eintauchen!

3.8 Vordefinierte Klassen und Packages

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 *Packages* nennen. Im neuen Java Standard 2 (den dieses Buch behandelt) gibt es Dutzende von Packages mit weit über 1000 Klassen! Eine immense Zahl, nicht wahr? Alle werden wir im Laufe des Buches nicht kennen lernen, aber die wichtigsten und nützlichsten. Danach werden Sie als alter Java-Hase kein Problem mehr haben, in der Java-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: Man *importiert* die Klassen durch Angabe des Package-Namens.

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

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

Meistens braucht man mehrere Klassen aus einem Package. Anstatt nun jede einzelne Klasse explizit zu importieren, kann man auch das ganze Package importieren:

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

Der * ist also eine Art Platzhalter.

Mit Hilfe von `import` können nur die Klassen eines Package importiert werden, die im Package als `public` deklariert sind (siehe auch Abschnitt 5.3).

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 Package, 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 Package heißt! Genau. Aber für die anderen Packages gilt das oben Gesagte. Sie müssen die einzelnen Klassen oder das jeweilige gesamte Package explizit importieren.

Mittlerweile besitzen wir auch das Hintergrundwissen, um Aufrufe wie

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

besser zu verstehen. `System` ist eine Klasse aus dem Paket `java.lang` (weswegen keine `import`-Anweisung nötig ist, um auf `System` zuzugreifen). Wenn Sie in der Java-Referenz (kann über das Hilfe-Menü des JBuilder aufgerufen oder – falls Ihre JBuilder-Version ohne Java-Referenz ausgeliefert wurde – von der Sun-Website heruntergeladen werden) unter `java.lang` und danach unter `System` nachschlagen, sehen Sie, dass `out` eine statische Instanzvariable von `System` ist. Da `out` als `static` deklariert ist, braucht man für den Zugriff auf `out` kein Objekt der Klasse `System`, sondern kann direkt über den Klassennamen zugreifen: `System.out`.

Die statische Instanzvariable `out` ist selbst ein Klassenobjekt (der Klasse `PrintStream`), das über die Methode `println()` verfügt, die wir zur Ausgabe von Texten verwenden können.

`System.out.println` ruft also über das statische Objekt `System.out` die Methode `println` auf.

Für statische Instanzvariablen gibt es eine eigene Bezeichnung: man nennt sie Klassenvariablen, siehe Abschnitt 5.3.

3.9 Test

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

2. Welche der folgenden Variablennamen sind nicht zulässig?

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

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
```

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

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

5. Die Division haben Sie kennen gelernt. Erinnern Sie sich noch an den Unterschied zwischen

```
double y;
y = 250/4;
```

und

```
double y;
y = 250.0/4;
```

6. Warum rechnet der Computer mit Binärzahlen?

7. Was sind Klassen?

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

9. Angenommen, Sie wollten einen Flugsimulator schreiben. Ihre Szenerie ist ganz einfach aufgebaut und besteht praktisch nur aus einem Untergrund und zwei Hochhäusern, die umflogen werden sollen. Überlegen Sie sich, welche Klassen Sie für diesen Flugsimulator definieren müssen,

welche Eigenschaften und Methoden benötigt werden und welche Instanzen Sie erzeugen müssen.

10. 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?

11. Was passiert in obiger Aufgabe, wenn Sie die Methode `bremsen` aufrufen, bevor Sie die Methoden `anlassen` und `beschleunigen` aufgerufen haben?

12. Welches Package muss nicht explizit importiert werden?

3.10 Lösungen

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

2. 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. Die folgenden Deklarationen sind nicht zulässig:

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

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. 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. 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 binärkodiert.

7. 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. Klassen sind Datentypen, Instanzen sind Variablen von Klassen.

9. Die erste Klasse dient zur Beschreibung des Flugzeugs. Als Eigenschaften sollten Sie zumindest zwei Gruppen von Instanzvariablen deklarieren:

   - ✘ eine Gruppe, die das Flugzeug beschreibt (Flugzeugtyp, Anzahl Turbinen, Spannweite, Höchstgeschwindigkeit etc.)
   - ✘ eine Gruppe, die den aktuellen Zustand des Flugzeugs beschreibt (Position, Flughöhe, Geschwindigkeit etc.)

   Als Methoden brauchen Sie wenigstens `Beschleunigen()` und `Bremsen()`.

```
//Hilfsklasse für Positionsangaben
class CKoord
  {
  int m_x;
  int m_y;

  CKoord(int x, int y)
    {
    m_x = x;
    m_y = y;
    }
  }

//Flugzeugklasse
class CFlugzeug
  {
  String m_typ;
  int    m_turbinen;
  int    m_spannweite;
  int    m_maxGeschw;
```

```
  CKoord  m_position;
  int    m_hoehe;
  int    m_aktGeschw;

  CFlugzeug(String typ, int turb, int weite,
            int geschw)
    {
    m_typ = typ;
    m_turbinen = turb;
    m_spannweite = weite;
    m_maxGeschw = geschw;
    m_position = new CKoord(0,0);
    m_hoehe = 0;
    m_aktGeschw = 0;
    }

  int beschleunigen()
    {
    // Akt_Geschw erhöhen, Höhe und Position ändern
    return m_aktGeschw;
    }

  int bremsen()
    {
    // Akt_Geschw senken, Höhe und Position ändern
    return m_aktGeschw;
    }
  }
```

Für die Hochhäuser brauchen Sie vor allem Instanzvariablen für die Position sowie Breite, Höhe und Tiefe. Wenn Sie möchten, können Sie auch eine Methode vorsehen, die aufgerufen wird, wenn ein Flugzeug in ein Hochhaus kracht.

```
// Klasse für Hochhäuser
class CHochhaus
  {
  CKoord m_position;
  int m_breite;
  int m_hoehe;
  int m_tiefe;
```

```
  CHochhaus(CKoord pos, int b, int h, int t)
    {
    m_position = pos;
    m_breite = b;
    m_hoehe = h;
    m_tiefe = t;
    }
  }
```

Für jedes Flugzeug, das in Ihrer Szenerie herumfliegt, bilden Sie eine eigene Instanz.

Da es zwei Hochhäuser gibt, benötigen Sie zwei Instanzen der Klasse `CHochhaus`.

Für den Boden brauchen Sie im Prinzip keine spezielle Klasse.

```
public class CFlugzeugSimulator
  {
  public static void main(String[] args)
    {
    CFlugzeug meines =
       new CFlugzeug("Sportflugzeug",1,5,300);
    CHochhaus haus1 =
       new CHochhaus(new CKoord(50,100),40,120,40);
    CHochhaus haus2 =
       new CHochhaus(new CKoord(150,80),30,140,40);

    meines.beschleunigen();
    meines.bremsen();
    }
  }
```

10. Zuerst müssen Sie die Klasse `CAuto` importieren. Unter der Annahme, dass die Klasse `CAuto` als `public` deklariert ist und keinem besonderen Package angehört, kopieren Sie die class-Datei in das Verzeichnis Ihres Programmes und setzen die Anweisung

    ```
    import CAuto;
    ```

    an den Anfang Ihres Quelltextes. Dann bilden Sie eine Instanz der Klasse:

    ```
    CAuto meinAuto = new CAuto();
    ```

    und rufen bei Bedarf einfach die Methode `anlassen()` auf:

    ```
    meinAuto.anlassen();
    ```

11. Eigentlich sollte es nichts ausmachen, wenn die Methode `bremsen()`, vor den Methoden `anlassen()` und `beschleunigen()` aufgerufen wird. Was aber wirklich passiert, hängt natürlich von der Implementierung ab, die Ihr Freund vorgesehen hat.

    Nehmen wir an, die Methode `bremsen()` reduziert die aktuelle Geschwindigkeit um 10 km/h. Dann kann es passieren, dass ein Aufruf der Methode `bremsen()` vor dem Aufruf der Methode `beschleunigung()` die aktuelle Geschwindigkeit auf -10 km/h zurücksetzt, einen negativen und somit ungültigen Wert. Gemäß den Regeln der Objektorientiertheit sollte die Klasse selbst dafür sorgen, dass ihre Daten (Instanzvariablen) nur vernünftige Werte annehmen. Die Methode `bremsen()` sollte also ständig die aktuelle Geschwindigkeit kontrollieren und diese nicht unter 0 km/h herabsetzen.

    Auf diese Weise sorgt die interne Implementierung der Klasse dafür, dass die Integrität ihrer Daten erhalten bleibt und Programmierfehler durch unsachgemäßen Gebrauch der Klasse weitestgehend verhindert werden. Man bezeichnet dies auch als Information hiding oder Kapselung.

12. Das Package `java.lang` braucht nicht explizit importiert zu werden.