

Inhalt

Vorwort	XIII
1 Bevor es losgeht	1
1.1 Was ist Java? – Teil I	1
1.2 Was ist ein Programm?	4
1.3 Wie werden Programme erstellt?	5
1.4 Von Compilern und Interpretern	6
1.5 Was ist Java? – Teil II	7
1.6 Vorbereitung zum Programmieren	9
2 Der erste Kontakt	17
2.1 Die erste Java-Anwendung	17
2.2 Zusammenfassung	25
2.3 Fragen und Antworten	26
2.4 Übungen	26
3 Von Daten, Operatoren und Objekten	27
3.1 Variablen und Anweisungen	27
3.2 Operatoren	37
3.3 Typumwandlung	40
3.4 Objekte und Klassen	44
3.5 Arrays	57
3.6 Vordefinierte Klassen und Pakete	60
3.7 Zusammenfassung	62
3.8 Fragen und Antworten	62
3.9 Übungen	64

4	Programmfluss und Fehlererkennung mit Exceptions	65
4.1	Die Axiomatik des Programmablaufs	65
4.2	Modularisierung durch Klassen und Methoden	66
4.3	Kontrollstrukturen	79
4.4	Fehlerbehandlung durch Exceptions	92
4.5	Zusammenfassung	98
4.6	Fragen und Antworten	99
4.7	Übungen	101
5	Objektorientierte Programmierung mit Java	103
5.1	Vererbung	103
5.2	Methoden (Klassenfunktionen)	114
5.3	Variablen- und Methodensichtbarkeit	120
5.4	Innere Klassen	129
5.5	Mehrfachvererbung und Schnittstellen	130
5.6	Zusammenfassung	134
5.7	Fragen und Antworten	135
5.8	Übungen	137
6	Ein- und Ausgabe	139
6.1	Streams	139
6.2	Ausgaben auf den Bildschirm	140
6.3	Ausgabe in Dateien	147
6.4	Eingaben von Tastatur	149
6.5	Aus Dateien lesen	153
6.6	Ein wichtiger Punkt: korrekte Exception-Behandlung	156
6.7	Rund um Strings	158
6.8	Zusammenfassung	167
6.9	Fragen und Antworten	167
6.10	Übungen	168
7	Collections und weitere nützliche Klassen	171
7.1	Zufallszahlen erzeugen	171
7.2	Zeit- und Datumsangaben	173
7.3	Zeichenfolgen zerlegen	177
7.4	Komplexe Datenstrukturen (Collections)	178
7.5	Algorithmen	191

7.6	Zusammenfassung	193
7.7	Fragen und Antworten	193
7.8	Übungen	194
8	Grundlagen der GUI-Programmierung	195
8.1	Der GUI-Reiseführer	196
8.2	Aufbau einer GUI-Anwendung	197
8.3	Das Ereignis-Modell des AWT	203
8.3.1	java.awt.event importieren	206
8.3.2	Ereignislauscher definieren	207
8.3.3	Lauscher für Quelle registrieren	209
8.3.4	Adapter	210
8.3.5	Einige abschließende Anmerkungen	211
8.4	Chamäleon sein mit UIManager und Look&Feel	214
8.5	Ein umfangreicheres Beispiel	216
8.6	Zusammenfassung	217
8.7	Fragen und Antworten	217
8.8	Übungen	219
9	Grafik, Grafik, Grafik	221
9.1	Das Arbeitsmaterial des Künstlers	221
9.2	Erweitertes Layout mit Panel-Containern	232
9.3	Kreise, Rechtecke und Scheiben	235
9.4	Freihandlinien	240
9.5	Noch mehr Grafik mit Java2D	245
9.6	Zusammenfassung	250
9.7	Fragen und Antworten	251
9.8	Übungen	251
10	Bilder, Bilder, Bilder	253
10.1	Der Bildbetrachter	253
10.2	Dateien öffnen und speichern: die Klasse JFileChooser	259
10.3	Laden und Anzeigen von Bildern	261
10.4	Zusammenfassung	266
10.5	Fragen und Antworten	266
10.6	Übungen	266

11	Text, Text, Text	267
11.1	Ein Texteditor	267
11.2	Umgang mit Text: JTextField, JTextArea und JTextPane	268
11.3	Kombinationsfelder	276
11.4	Eigene Dialoge	279
11.5	Nach Textstellen suchen	284
11.6	Unterstützung der Zwischenablage	287
11.7	Drucken	290
11.8	Zusammenfassung	292
11.9	Fragen und Antworten	292
11.10	Übungen	293
12	Menüs und andere Oberflächenelemente	295
12.1	Die Komponentenhierarchie	296
12.2	Die Basisklasse Component	296
12.3	Statische Textfelder (JLabel)	298
12.4	Schaltflächen (JButton)	300
12.5	Eingabefelder (JTextField und JTextArea)	302
12.6	Optionen (JCheckBox, JRadioButton)	305
12.7	Listen- und Kombinationsfelder (JList und JComboBox)	308
12.8	Bildlaufleisten (JScrollBar)	311
12.9	Menüleisten (JMenuBar)	312
12.10	Zusammenfassung	315
12.11	Fragen und Antworten	316
12.12	Übungen	317
13	Threads und Animation	321
13.1	Multithreading mit Java	321
13.2	Eigene Threads erzeugen: die Klasse Thread	325
13.3	Eigene Threads erzeugen: die Runnable-Schnittstelle	329
13.4	Wissenswertes rund um Threads	331
13.5	Threads und Animation I	334
13.6	Threads und Animation II	339
13.6.1	SwingWorker	340
13.7	Zusammenfassung	345
13.8	Fragen und Antworten	345
13.9	Übungen	346

14	Sound	347
14.1	Was ist eine URL?	347
14.2	Sounddateien abspielen	349
14.3	Wiedergabe von MP3	350
14.4	Tonerzeugung mit MIDI	351
14.4.1	Abspielen einer MIDI-Datei	351
14.4.2	Selber Musik machen	352
14.5	Zusammenfassung	354
14.6	Fragen und Antworten	355
14.7	Übungen	355
15	Die Datenbankschnittstelle JDBC	357
15.1	Datenbanken-ABC	358
15.2	Die JDBC-Schnittstelle	360
15.3	Vorbereitung für JavaDB	361
15.4	Zugriff auf eine Datenbank	361
15.4.1	Verbindungsaufbau	361
15.4.2	Lese- und Schreiboperationen durchführen	363
15.4.3	Verbindung schließen	365
15.5	Zusammenfassung	369
15.6	Fragen und Antworten	369
15.7	Übungen	370
16	Was wir noch erwähnen wollten	371
16.1	Aufzählungen (enum)	371
16.1.1	Definition	372
16.1.2	Variablen definieren	372
16.1.3	Aufzählungskonstanten vergleichen	372
16.1.4	Aufzählungen und switch	373
16.1.5	Aufzählungen und for	374
16.2	Lambda-Ausdrücke	375
16.3	Java Generics	376
16.3.1	Einleitung	376
16.3.2	Syntax	378
16.3.3	Eingeschränkte Platzhalter	379
16.3.4	Parameter und Variablen von generischen Typen	380

16.4	Debuggen	381
16.4.1	Grundsätzliches Vorgehen	382
16.4.2	Der Debugger JDB	383
16.5	Anwendungen weitergeben	384
16.5.1	Ohne JRE geht es nicht	384
16.5.2	Java-Anwendungen ausführen: von .class bis .exe	385
Anhang A: Lösungen		389
Anhang B: Installation des JDK		409
B.1	Installation	410
B.2	Anpassen des Systems	412
B.2.1	Erweiterung des Systempfads	412
B.2.2	Installation testen	415
B.2.3	Setzen des Klassenpfads	416
B.3	Die Java-Dokumentation	417
B.4	Wo Sie weitere Hilfe finden	418
Anhang C: Schlüsselwörter		419
Anhang D: Syntaxreferenz		421
D.1	Grundgerüste	421
D.2	Datentypen	422
D.3	Konstanten	423
D.4	Variablen	423
D.5	Operatoren	424
D.5.1	Arithmetische Operatoren	424
D.5.2	In- und Dekrement	425
D.5.3	Relationale Operatoren	426
D.5.4	Logische Operatoren	427
D.5.5	Bitweise Operatoren	427
D.5.6	Typumwandlung (Cast)	428
D.5.7	instanceof	428
D.5.8	Der Bedingungsoperator	428
D.5.9	Reihenfolge der Operatorenauswertung	429
D.6	Strings	431

D.7	Ablaufsteuerung	432
D.7.1	Einfache Verzweigung	432
D.7.2	if-else-Verzweigung	432
D.7.3	switch-Verzweigung	433
D.7.4	while-Schleifen	433
D.7.5	do-while-Schleifen	433
D.7.6	for-Schleife	433
D.7.7	Abbruchbefehle	434
D.8	Aufzählungen	434
D.9	Arrays	435
D.9.1	Einfache Arrays	435
D.9.2	Mehrdimensionale Arrays	437
D.10	Klassen	437
D.10.1	Definition	437
D.10.2	Felder	441
D.10.3	Methoden	441
D.10.4	Konstruktoren	443
D.10.5	Instanzbildung	444
D.11	Vererbung	444
D.12	Abstrakte Klassen	446
D.13	Schnittstellen	447
D.14	Lambda-Ausdrücke	448
D.15	Generika	449
D.15.1	Generische Klassen	449
D.15.2	Generische Schnittstellen	449
D.15.3	Generische Methoden	450
D.15.4	Eingeschränkte Platzhalter	450
D.15.5	Arrays von generischen Typen	451
D.15.6	Vererbung	451
D.16	Ausnahmebehandlung	452
D.16.1	Grundmodell	452
D.16.2	Fehlerbehandlung mit mehreren catch-Blöcken	452
D.16.3	Fehlerbehandlung mit finally-Block	452
D.16.4	Exceptions auslösen	453
D.17	Stilkonventionen für Bezeichner	453

Anhang E: Java-Klassenübersicht	455
E.1 java.io	455
E.2 java.lang	456
E.3 java.applet	456
E.4 java.awt	457
E.5 java.awt.event	458
E.6 java.awt.geom	459
E.7 java.net	459
E.8 java.sql	460
E.9 javax.sound.midi	460
E.10 javax.swing	461
E.11 java.util	462
 Anhang F: Literatur und Adressen	 463
F.1 Bücher	463
F.2 Zeitschriften	464
F.3 Ressourcen im Internet	465
 Anhang G: Die DVD zum Buch	 467
 Index	 469

Vorwort

Dieses Buch soll Sie auf leicht verständliche und gleichsam unterhaltsame Weise in die Programmierung mit Java einführen. Vorhandene Programmierkenntnisse können von Vorteil sein, werden aber nicht vorausgesetzt. Schritt für Schritt werden Sie sich in Java einarbeiten und erfahren, wie Sie die Mächtigkeit der Sprache für Ihre Zwecke nutzen können.

Sie werden Ihre ersten Java-Anwendungen schreiben, sich mit der objektorientierten Programmierung vertraut machen und erfahren, wie man in Java Anwendungen mit grafischen Benutzeroberflächen programmiert. In den letzten Kapiteln des Buchs wenden wir uns dann noch diversen Fortgeschrittenenthemen zu, wie der Implementierung von Threads und Animationen oder der Datenbankunterstützung.

Am Ende eines jeden Kapitels finden Sie eine Reihe von Testfragen und eine Zusammenfassung des behandelten Stoffs sowie einige Übungsaufgaben, die zur Vertiefung und Wiederholung des Stoffs dienen, vor allem aber auch die Lust am Programmieren anregen sollen.

Programmierkenntnisse sind wie erwähnt nicht erforderlich, aber Sie sollten auch nicht zu den Zeitgenossen gehören, die bisher noch jedem Computer erfolgreich aus dem Weg gegangen sind. Spaß am Programmieren können wir Ihnen nur dann vermitteln, wenn Sie selbst ein wenig guten Willen und Ausdauer mitbringen. Gute Laune, eine Portion Neugier, dieses Buch – was brauchen Sie noch?

Um mit Java programmieren zu können, benötigen Sie ein entsprechendes Entwicklungspaket, das sogenannte JDK (Java Development Kit). Optional können Sie zusätzlich zu dem JDK eine integrierte Entwicklungsumgebung wie z. B. NetBeans oder die Open-Source-Software Eclipse verwenden.

Für Java-Einsteiger ist der Einsatz einer integrierten Entwicklungsumgebung allerdings häufig recht verwirrend und lenkt vom eigentlichen Primärziel, die Programmiersprache Java zu lernen, unnötig ab. Schlimmer noch: Die meisten Entwicklungs-

umgebungen sind für fortgeschrittene Programmierer konzipiert und erleichtern deren tägliche Programmierarbeit, indem sie komplexe Arbeitsschritte automatisieren, vordefinierte Codegerüste anbieten, eigenständig Code erzeugen. Für Anfänger ist dies ein Desaster! Nehmen wir nur einmal das Codegerüst einer einfachen Konsolenanwendung, das in Java aus ungefähr vier bis sieben Zeilen Code besteht. Wenn Ihnen dieses Codegerüst stets von Ihrer Entwicklungsumgebung fertig vorgelegt wird, werden Sie sich kaum die Mühe machen, es je selbst einmal abzutippen. Sie werden es sich vielleicht anschauen und versuchen, es zu verstehen, aber Sie werden es nie wirklich verinnerlichen. Wenn Sie dann später einmal an einem Rechner arbeiten müssen, auf dem keine Entwicklungsumgebung installiert ist, werden Sie mit Schrecken feststellen, dass es Ihnen unmöglich ist, das Grundgerüst aus dem Kopf nachzustellen. Nun, ganz so schlimm wird es vielleicht nicht kommen, aber der Punkt ist, dass die Annehmlichkeiten der Entwicklungsumgebungen den Anfänger schnell dazu verführen, sich mit zentralen Techniken und Prinzipien der Java-Programmierung nur oberflächlich auseinanderzusetzen.

Aus diesem Grund legen wir in diesem Buch Wert auf Handarbeit mit elementarsten Mitteln: Wir setzen unsere Quelltexte in einem einfachen Texteditor auf, wandeln die Quelltexte mithilfe des Java-Compilers `javac` aus dem JDK in ausführbare Programme um und führen diese dann mit dem Java-Interpreter `java` (ebenfalls im JDK enthalten) aus. Wie Sie dabei im Einzelnen vorgehen und was Sie beachten müssen, erfahren Sie in den einleitenden Kapiteln und im Anhang dieses Buchs.

`www.carpelibrum.de`

Falls Sie während der Buchlektüre auf Probleme oder gar auf inhaltliche Fehler stoßen, sollten Sie nicht zögern, uns eine E-Mail unter Angabe von Buchtitel und Auflage zu senden. Allerdings schauen Sie bitte zuerst auf unserer Buchseite *www.carpelibrum.de* nach, ob sich nicht dort schon eine Antwort findet. Neben Aktualisierungen, Fehlerkorrekturen und Antworten auf typische Fragen finden Sie dort auch Hinweise auf weitere Bücher rund ums Thema Programmieren.

Viel Erfolg mit Java wünschen Ihnen

Dirk Louis (*autoren@carpelibrum.de*)

Peter Müller (*leserfragen@gmx.de*)

Saarbrücken, im Sommer 2014

3

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 Basic oder C gilt, so ist es erst recht wahr für Java mit seinem konsequent objekt-orientierten 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 uns die aufregende Welt der Java-Programmierung Stück für Stück erobern und unserer eigenen Kreativität als Programmierer Tür und Tor öffnen.

■ 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 Mathematikunterricht 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 im Binärformat speichert.

Für die Anwender von Computern ist dies natürlich nicht zumutbar und die Computerrevolution – 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 Größte 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 ErstesBeispiel {
    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 Kapitel 2.1 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.



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.



MERKSATZ

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

```
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;
```

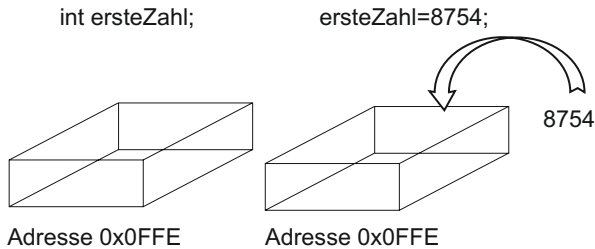


Bild 3.1
Deklaration und Zuweisung

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.



ACHTUNG!

= bedeutet Zuweisung.

== bedeutet Vergleich.

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 ErstesBeispiel.java

```
public class ErstesBeispiel {
    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!

Leser mit Vorkenntnissen in Sprachen wie C++ 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 verschwommen, denn die Deklaration einfacher Variablen schließt auch in C++ 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 hinten 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.

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
<code>boolean</code>	boolescher Wert (wahr, falsch)	<code>true</code> , <code>false</code>
<code>char</code>	Zeichen, Buchstabe	Unicode-Werte
<code>byte</code>	ganze Zahl	−128 bis +127
<code>short</code>	ganze Zahl	−32768 bis 32767
<code>int</code>	ganze Zahl	−2.147.483.648 bis +2.147.483.647
<code>long</code>	ganze Zahl	−9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
<code>float</code>	Fließkommazahl	−3,40282347*1038 bis +3,40282347*1038
<code>double</code>	Fließkommazahl	−1,7976931348623157*10308 bis +1,7976931348623157*10308

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.

**ACHTUNG!**

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`.



Klein ist nicht gleich klein!

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 Buchstaben¹, '_' 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. In Anhang C finden Sie eine Liste der reservierten Wörter.

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.



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 (die sogenannte *CamelCase*-Schreibweise). 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 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 sogenannten *Literals*.

```
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 für 63 0xFF1F // hexadezimal für 65311 0b00010001 // binär für 17 0b0001_0001 // binär mit eingefügtem // Unterstrich zur // besseren Lesbarkeit

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

**ACHTUNG!**

Ab Java 7 kann man die Lesbarkeit von Zahlenliteralen durch das Einfügen von Unterstrichen verbessern, also z.B. 1_000_000 statt 1000000. Die Unterstriche dürfen aber nur **zwischen** die Ziffern gesetzt werden.

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 Operatoren.java

```
public class Operatoren {  
    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 sieht 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 Hinzusaddieren 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
<code>++, --</code>	Inkrement, Dekrement	Erhöht oder erniedrigt den Wert einer Variablen um 1.
<code>!</code>	logisches NICHT	Negiert den Wahrheitswert einer Aussage (beispielsweise eines Vergleichs). Wird meist in Kontrollstrukturen (siehe Kapitel 4) verwendet.
<code>*, /</code>	Multiplikation, Division	Multiplikation und Division
<code>%</code>	Modulo-Division	Liefert den Rest einer ganzzahligen Division. 4 % 3 liefert z. B. 1.
<code>-, +</code>	Subtraktion, Addition	Subtraktion und Addition
<code><=, <, >, >=</code>	Vergleich	Zum Vergleich zweier Werte. Die Operatoren liefern <code>true</code> oder <code>false</code> zurück.
<code>==, !=</code>	Vergleich (gleich, ungleich)	Zum Vergleich auf Gleichheit oder Ungleichheit. Die Operatoren liefern <code>true</code> oder <code>false</code> zurück.
<code>&&</code>	logisches UND	Verknüpft zwei Aussagen. Liefert <code>true</code> , wenn beide Aussagen <code>true</code> sind. <code>if ((x < 1) && (y > 1))</code>
<code> </code>	logisches ODER	Verknüpft zwei Aussagen. Liefert <code>true</code> , wenn eine der beiden Aussagen <code>true</code> ist. <code>if ((x < 1) (y > 1))</code>

Tabelle 3.3 Operatoren (Fortsetzung)

Operator	Beschreibung	Beispiel
&	bitweises UND	UND-Verknüpfung der Binärpräsentation zweier Zahlen <pre>var1 = 1; // ...0001 var2 = 5; // ...0101 var3 = var1 & var2; // ...0001</pre>
	bitweises ODER	ODER-Verknüpfung der Binärpräsentation zweier Zahlen <pre>var1 = 1; // ...0001 var2 = 5; // ...0101 var3 = var1 var2; // ...0101</pre>

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 Demo1 {  
    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 Demo2 {  
    public static void main(String[] args) {  
        int x;  
        double y = 3.14;  
  
        x = y;  
        System.out.println(x);  
    }  
}
```

In diesem Fall 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 Demo3 {  
    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 60 000 und das ist außerhalb des Wertebereichs von `short`! Das Ergebnis lautet in diesem Fall -5536.

Wie kommt dieses merkwürdige Ergebnis zustande?

Als Integer-Wert wird 60 000 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 Demo4 {  
    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 unter Umständen 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 Programmieretechnik.

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 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 zusammenge-

hörigen 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 `Mitarbeiter`.

```
class Mitarbeiter {  
}
```



Unter Java-Programmierern ist es üblich, Klassennamen mit einem Großbuchstaben beginnen zu lassen, danach wird klein weitergeschrieben. In zusammengesetzten Namen beginnt jeder Teil mit einem Großbuchstaben.

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 Mitarbeiter {  
    String m_name;  
    String m_vorname;  
    int    m_gehalt;  
}
```

Langsam nimmt unsere Klasse – und damit die Mitarbeiter-Objekte, die wir aus ihr erzeugen werden – 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 Klassentyp, genau wie unsere Klasse `Mitarbeiter`. 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.



Strings = Zeichenketten

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

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*²) 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 *Mitarbeiter* implementiert:

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

    Mitarbeiter(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);
    }
}
```

² Wir haben uns für das *m* als Präfix entschlossen, weil es im Schriftbild etwas unaufdringlicher ist als das *f* für Feld.


```
        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 `Mitarbeiter` besitzt nun drei Methoden mit den Namen `Mitarbeiter`, `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:

```
methodName() {  
    Anweisungen;  
}
```



Mittlerweile haben wir die dritte Art von *Bezeichnern* (Namen, die der Programmierer einführt und per Deklaration dem Compiler bekanntgibt) 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 einen 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.

Was 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 Kapitel 5.2 werden wir dann noch sehen, wie mithilfe des Schlüsselworts `return` dieser Rückgabewert an den Aufrufer der Methode zurückgeliefert wird.

Eine vollständige Methodendeklaration würde jetzt folgendem Schema folgen:

```
Rückgabotyp methodName(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).

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:

```
Mitarbeiter(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).



MERKSATZ

Jede Klasse braucht zumindest einen Konstruktor zur Initialisierung ihrer Felder. Wenn Sie selbst keinen solchen Konstruktor vorsehen, weist der Compiler der Klasse einen Ersatzkonstruktor 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 Methoden³ hat es sich eingebürgert, aktive Namen (mit Verben) zu wählen und sie mit Kleinbuchstaben beginnen zu lassen. Zusätzlich lässt man sinnvolle Teilwörter mit Großbuchstaben beginnen, z. B. *leseKonfigurationsDaten()*.

³ Für Konstruktoren hat man natürlich keine Wahl bei der Namensgebung, da hier der Klassenname genommen werden muss.

Instanzen

Moment mal, Instanzen? Was soll denn das sein? Denken Sie am besten an den Mitarbeiter aus der realen Welt, nachdem wir die Klasse `Mitarbeiter` modelliert haben. Die Klasse `Mitarbeiter` 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 `Mitarbeiter`!

Anders ausgedrückt: Unsere Klasse `Mitarbeiter` 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. In Kapitel 5.3, Abschnitt „Dreierlei Variablen“ werden wir daher die wichtigsten Begriffe noch einmal zusammenfassen und gegenüberstellen.

Mit Klassen programmieren

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

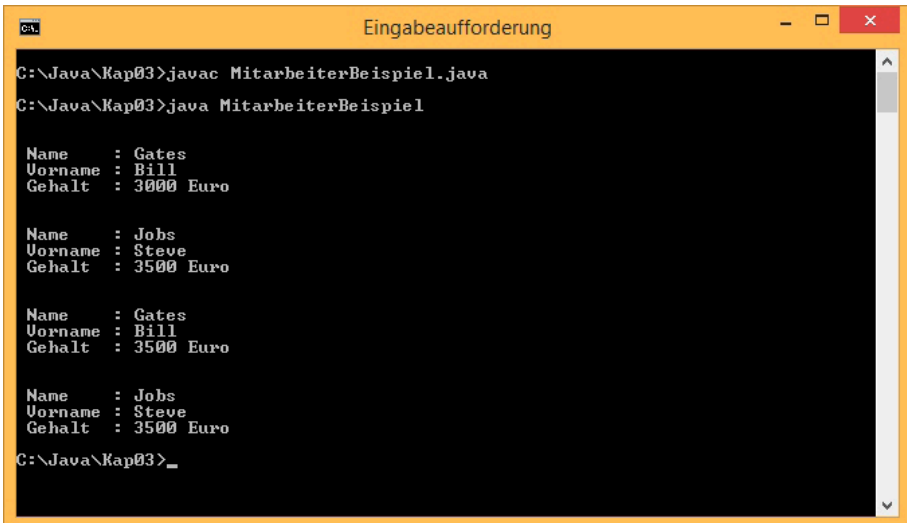
Listing 3.3 `MitarbeiterBeispiel.java`

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

    Mitarbeiter(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 MitarbeiterBeispiel {  
    public static void main(String[] args) {  
        // 2 neue Mitarbeiter instanzieren  
        Mitarbeiter billy = new Mitarbeiter("Gates","Bill",3000);  
        Mitarbeiter stevie = new Mitarbeiter("Jobs","Steve",3500);  
  
        // Daten ausgeben  
        billy.datenAusgeben();  
        stevie.datenAusgeben();  
  
        // Gehalt von a erhöhen  
        billy.gehaltErhoehen(500);  
  
        // Kontrolle  
        billy.datenAusgeben();  
        stevie.datenAusgeben();  
    }  
}
```



```
C:\Java\Kap03>javac MitarbeiterBeispiel.java  
C:\Java\Kap03>java MitarbeiterBeispiel  
  
Name      : Gates  
Vorname   : Bill  
Gehalt    : 3000 Euro  
  
Name      : Jobs  
Vorname   : Steve  
Gehalt    : 3500 Euro  
  
Name      : Gates  
Vorname   : Bill  
Gehalt    : 3500 Euro  
  
Name      : Jobs  
Vorname   : Steve  
Gehalt    : 3500 Euro  
C:\Java\Kap03>_
```

Bild 3.2 Ausgabe des Programms MitarbeiterBeispiel

Sie sollten das Beispiel in den Editor eingeben, kompilieren und ausführen. Denken Sie daran, dass die Datei den gleichen Namen wie die `public`-Klasse tragen muss, also in diesem Fall *MitarbeiterBeispiel.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 `Mitarbeiter` angelegt:

```
Mitarbeiter billy = new Mitarbeiter("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 `Mitarbeiter` erzeugen soll. Dazu wird mithilfe des Schlüsselworts `new` der Konstruktor der Klasse aufgerufen, der drei Parameter erwartet, die wir ihm ordnungsgemäß übergeben.



MERKSATZ

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

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

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 Bild 3.3).

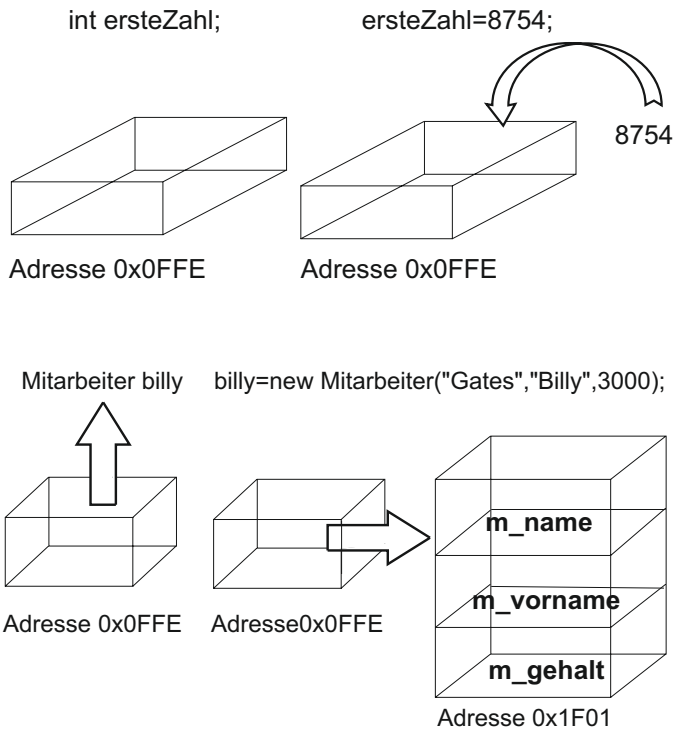


Bild 3.3 Instanzbildung

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 `Mitarbeiter` 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 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:

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


Zuerst wird eine Variable vom Typ `Mitarbeiter` 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 ...

**ACHTUNG!**

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 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 `Mitarbeiter` muss also immer zuerst zwei Zeichenfolgen 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 `Mitarbeiter` 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];
Mitarbeiter[] personalliste = new Mitarbeiter[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“.



ACHTUNG!

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 Mitarbeiter("Schramm", "Herbert", 3500);
personalliste[1] = billy; // billy sei eine Mitarbeiter-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 beschert 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 `ArrayIndexOutOfBoundsException`.



ACHTUNG!

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 6.7 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 vielen 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:

```
Mitarbeiter[] personalliste = new Mitarbeiter[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 im 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 aktuellen Java-Standard (den dieses Buch behandelt) gibt es Dutzende von Paketen mit weit über 1000 Klassen! Eine immense Zahl, nicht wahr? Alle werden wir im Laufe des Buchs nicht kennenlernen, aber die wichtigsten und nützlichsten. Danach werden Sie als alter Java-Hase kein Problem mehr haben, in der Java-API-Dokumentation unter <http://docs.oracle.com/javase/8/docs/api/> 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:

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

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

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 bekanntzumachen. 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 `Auto` mit den Feldern `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?

Index

A

- Abbruchbefehle 434
- abstract 111, 439
- AbstractButton (Klasse) 306
- Action (Klasse) 288
- ActionEvent (Klasse) 208, 232, 257
- ActionListener 208, 257, 282, 288
- actionPerformed() 208, 232
- addActionListener() 209
- addItemListener() 278
- addMouseListener() 400
- addMouseMotionListener() 404
- Adressen 463
- Algorithmen 191
 - Sortieren (Arrays) 191
 - Sortieren (Listen) 192
 - Suchen (Arrays) 192
 - Suchen (Listen) 192
- Animationen 334
 - Ablauf 339
- Anweisungen 22, 35
- Anwendungen
 - an Freunde weitergeben 9, 384
 - beenden 211
 - erstellen 18
 - Hauptklasse 75
 - Programmablauf 65
 - Programmgerüst 17, 199
 - Programmstart 24
- API-Dokumentation 417
- Applets 3
- Arbeitsspeicher 30
- Arrays 57
 - als Parameter 118
 - Deklaration 57
 - in Schleifen bearbeiten 87
- Arrays (Klasse) 191
- ArrayList (Klasse) 181
- AudioClip (Schnittstelle) 349
- AudioInputStream (Klasse) 349
- Aufzählungen 84
- Aufzählungen (enum) 371
- Ausgabe *siehe* E/A:Ausgabe
- Autoboxing 182
- AWT 195
 - Ereignisbehandlung 203
 - Klassen
 - BorderLayout 399
 - Component 296
 - FlowLayout 201
 - Font 272
 - Graphics 224
 - GridBagLayout 318
 - GridLayout 233
 - Image 261
 - MouseAdapter 237
 - MouseMotionAdapter 214f.
 - Toolkit 209
 - WindowAdapter 211
 - Layout-Manager einrichten 201
- AWT-Event-Handling-Thread 323

B

- beep() 208
- Beispiele auf Buch-DVD 467
- Bezeichner 35
 - Variablen 36

Bilder 253
 – anzeigen 258
 – drawImage() 258
 – getImage() 261
 – in Bilder zeichnen 266
 – Klasse Image 261
 – Klasse ImageIcon 261
 – Klasse ImageIO 263
 – Klasse MediaTracker 264
 – Klasse Toolkit 264
 – laden 261, 263
 – speichern 263
 Bildlaufleisten 270, 311
 Binärdarstellung 27
 Bitoperationen 427
 BorderLayout (Klasse) 399
 break 84, 91
 Buch-DVD 467
 BufferedReader (Klasse) 155
 BufferedWriter (Klasse) 155
 ButtonGroup (Klasse) 236, 306
 Bytecode 7

C

C# 2
 C++ 2
 CamelCase 36
 case 83
 Casting 40, 42 *siehe* Typumwandlung
 catch 95
 class 46
 CLASSPATH-Umgebungsvariable 416
 Collections 178
 Collections (Klasse) 192
 Color (Klasse) 228
 Comparable (Schnittstelle) 191
 Compiler 5
 – javac 18
 – versus Interpreter 6
 Component (Klasse) 296
 Connection (Schnittstelle) 362
 console() 142
 Console (Klasse) 141, 149
 continue 91
 currentThread() 326

D

Dateien
 – Ausgabe in Datei 147
 – Bilddateien 261
 – Dateinamen abfragen 259
 – in GUI-Anwendungen 273
 – Klasse FileReader 154
 – Klasse FileWriter 148
 – Klasse PrintWriter 147
 – kopieren 395
 – Lesen aus Datei 153
 – öffnen 168
 Datenbanken 357
 – Datenbank 358
 – Datenbankanwendung 358
 – JDBC 360
 – relationale Datenbanksysteme 359
 – SQL 359, 363, 407
 – Tabellen 358
 – Treiber 360
 – Zugriff auf eine Datenbank 361
 Datenstrukturen 178
 – Hashtabellen 187, 289
 – Iteratoren 180
 – Keller 194
 – Liste 181
 – Menge 185
 – Sortieren (Arrays) 191
 – Sortieren (Listen) 192
 – Suchen (Arrays) 192
 – Suchen (Listen) 192
 – Wrapper-Klassen 182
 Datentypen 33
 – Arrays 57
 – enum 84
 – Klassen 46
 – umwandeln 40
 – Wertebereiche 34
 Datumsanzeige 173
 DBMS 358
 Debuggen 381
 DefaultListModel (Klasse) 310
 Definition 33 *siehe* Deklaration
 Deinstallation (JDK) 412
 Deklaration 32
 – Datentyp 30
 – versus Definition 32

delete() 149
 Dialoge 279
 – feste Größe 283
 – Klasse JDialog 282
 – Klasse JFileChooser 259
 – Klasse JOptionPane 286
 – modal/nichtmodal 282
 – nichtmodale 293
 – Schaltflächen 283
 – schließen 283
 – versus Fenster 280
 dispose() 243
 Document (Klasse) 275
 doInBackground() 340
 done() 340
 drawImage() 258
 Drucken 290
 – Klasse MessageFormat 291
 – print() 290
 DVD 467

E

E/A

– Ausgabe
 – auf Bildschirm 140
 – Console.printf() 142
 – FileWriter.write() 148
 – formatiert 142
 – in Dateien 147
 – Klasse Console 141
 – Klasse File 149
 – Klasse FileWriter 148
 – Klasse PrintStream 141
 – Klasse PrintWriter 145
 – System.out.print() 140
 – System.out.printf() 142
 – System.out.println() 140
 – Umlaute 141, 147
 – Eingabe
 – aus Datei 153
 – Console.readLine() 149
 – FileReader.read() 155
 – Klasse BufferedReader 155
 – Klasse BufferedWriter 155
 – Klasse Console 149
 – Klasse FileReader 154
 – Klasse Scanner 151
 – Scanner.nextInt() 153
 – Scanner.nextInt() 153
 – Scanner.nextLine() 153
 – von Tastatur 149
 – Streams 139
 Editoren 10
 – Editor (notepad) 10
 – Linux 10
 – Notepad++ 10
 – Windows 10
 EditorKits 288
 Einfügemarke 273
 Eingabe *siehe* E/A:Eingabe
 Eingabeaufforderung (Konsole) 12
 Eingabefelder 302
 Ein- und Ausgabe 139 *siehe* E/A
 else 79
 Endloschleifen 86
 Entwicklungsumgebungen
 – integrierte 14
 – selbst eingerichtete 10
 enum 84, 371
 Ereignisbehandlung 203
 – actionPerformed() 208, 232
 – Adapter-Klassen 210
 – MouseAdapter 237
 – MouseMotionAdapter 214, 242
 – WindowAdapter 211
 – addActionListener() 209
 – addItemListener() 278
 – addMouseListener() 400
 – addMouseMotionListener() 404
 – Ereignisquellen 204
 – Ereignisse 204
 – Event-Lauscher 204
 – für die Maus 400, 404
 – für Fenster 210, 211
 – für Kombinationsfelder 278
 – für Menübefehle 257, 288
 – für Schaltflächen 208
 – getActionCommand() 232
 – itemStateChanged() 279
 – Klassen importieren 206
 – Komponente ausgewählt 208
 – Lauscher
 – definieren 207
 – für mehrere Quellen 232
 – mit Quelle verbinden 209
 – Listener-Schnittstellen 207
 – Mausklicks 237

- Mauskoordinaten 238
- Maus überwachen 400
- mouseClicked() 400
- mouseDragged() 241
- MouseEvent-Parameter 238
- mousePressed() 238
- Parameter 208, 232
- Quelle identifizieren 232
- Schnittstellen
 - ActionListener 208, 257, 282, 288
 - FocusListener 213
 - ItemListener 278
 - KeyListener 213
 - MouseListener 237, 400
 - MouseMotionListener 241
 - TextListener 214
 - WindowListener 210
- windowClosing() 211
- Ziehen der Maus 241

Exceptions 93

- abfangen 93
- auslösen 94
- behandeln 95
- Code überwachen 95
- finally 156
- für Closeable-Objekte 157
- getMessage() 96
- printStackTrace() 96
- Schlüsselwort catch 95
- Schlüsselwort throw 94
- Schlüsselwort try 95

execute() 340

exists() 149

exit() 211

extends 106, 380

F

Farben 228

- definieren 229
- RGB-Format 229
- setColor() 228

Fehlerbehandlung 92 *siehe* Exceptions

Fehlersuche 381

Fenster 199

- anzeigen 200
- Dialoge 279
 - feste Größe 283

- Klasse JDialog 282
- modal/nichtmodal 282
- schließen 283

Ereignisbehandlung 210 f.

Fenstertitel 200

- feste Größe 283

- gestalten 200

- Hauptfenster 212

- instanzieren 200

- Layout wählen 201

- pack() 201

- Schaltflächen 283

- schließen 283

- setVisible() 200

- versus Dialoge 280

File (Klasse) 149

FileReader (Klasse) 154

final 37, 439

- Klassen 110
- Methoden 110
- Parameter 116
- Variablen 37

finalize() 123

finally 97, 156

FocusListener 213

Font (Klasse) 272

Fonts 272

- Font-Name 272
- installierte 273
- Klasse Font 272
- Namen 272
- Stil 272
- Systemfonts abfragen 404
- und Plattformabhängigkeit 272

for 85, 88, 181

Freihandlinien 240

G

gc() 124

Generics 179, 376

getAbsolutePath() 149, 261

getActionCommand() 232

getAvailableFontFamilyNames() 273, 404

getCanonicalPath() 149

getGraphics() 241, 243

getImage() (ImageIcon) 261

getMessage() 96

getName() 149

- Grafik 221
 - Animationen 334
 - Ablauf 339
 - Frame-by-Frame 339
 - Arbeitsmaterial 221
 - Bilddateien 253
 - Bilder anzeigen 258
 - drawImage() 258
 - getImage() 261
 - Klasse Image 261
 - Klasse ImageIcon 261
 - Klasse ImageIO 263
 - Klasse MediaTracker 264
 - Klasse Toolkit 264
 - laden (Anwendungen) 261
 - Farben 228, 229
 - Freihandlinien 240
 - Füllmuster 247
 - geometrische Figuren 238
 - getGraphics() 243
 - Gradientenfüllung 247
 - Graphics-Objekte selbst erzeugen 241
 - Hintergrundfarbe 227, 228
 - in Bilder zeichnen 266
 - in Komponenten 317
 - Java2D 245
 - JPanel 258
 - Klasse Graphics 224
 - Koordinatentransformationen 227
 - Leinwand 221
 - an Fenstergröße anpassen 258
 - Größe festlegen 227
 - Mausklicks 237
 - Neuzeichnen forcieren 223
 - paint() 222
 - paintComponent() 222, 258
 - Rekonstruktion 223
 - repaint() 223
 - Skalierung 228
 - Strichstärke 246
 - Ursprung verschieben 227
 - vorbereitende Maßnahmen 222
 - Vordergrundfarbe 227
 - Zeichenfläche 221
 - Zeichenkontext erzeugen 241
 - Zeichenmethoden 222, 224
 - Ziehen der Maus 241
 - zweidimensionale 245
- GraphicsEnvironment (Klasse) 404
- Graphics (Klasse) 224
- GridBagLayout (Klasse) 318
- GridLayout (Klasse) 233
- GUI (Graphical User Interface) 195
 - Aufbau 197
 - AWT 195
 - Bilddateien *siehe* Grafik:Bilddateien
 - Container 197, 200
 - Ereignisbehandlung 203 *siehe* Ereignisbehandlung
 - Adapter-Klassen 210
 - Ereignislauscher 204
 - Ereignisquellen 204
 - Klassen importieren 206
 - Lauscher definieren 207
 - Lauscher für mehrere Quellen 232
 - Lauscher mit Quelle verbinden 209
 - Listener-Schnittstellen 207
 - Maus überwachen 400
 - Parameter 232
 - Quelle identifizieren 232
 - Ereignislauscher 197
 - Fenster 199
 - anzeigen 200
 - Fenstertitel 200
 - gestalten 200
 - instanzieren 200
 - Grafiken *siehe* Grafik
 - Hauptfenster schließen 210
 - Klassen
 - JFrame 199
 - Klassen importieren 199
 - Komponenten 197
 - anordnen 201
 - Basisklasse Component 296
 - Bildlaufleisten (JScrollbar) 311
 - Bildlaufleisten (JScrollPane) 311
 - Eingabefelder (JPasswordField) 305
 - Eingabefelder (JTextField und JTextArea) 302
 - Hierarchie 296
 - Kombinationsfelder (JComboBox) 276, 308
 - Listenfelder (JList) 308
 - Menüleisten 257, 313
 - Optionsfelder 305
 - Schaltflächen (JButton) 200, 300
 - statische Textfelder (JLabel) 298
 - Titel ändern 399

- Layout-Manager 197, 201
- Menüleisten 257, 313 *siehe* Menüs
- Oberflächen 196
- Oberflächenelemente *siehe* Komponenten
- Programme beenden 210
- Programmgerüst 199
- Swing 195
- Text *siehe* Text
- Threads 340
- Überblick 196
- Gültigkeitsbereiche
- Anweisungsblock 424

H

- HashMap (Klasse) 188
- HashSet (Klasse) 185
- Hashtabellen 187, 289
- Hashtable (Klasse) 188
- Hauptfenster 212
- Hauptklasse 75

I

- if 79
- if-Bedingung 79
- Image (Klasse) 261
- ImageIcon (Klasse) 261, 317
- ImageIO (Klasse) 263
- immutable 158, 175
- implements 133
- import 60, 77
- Importieren
 - Swing-Klassen 199
- Initialisierung 32
- Inkrement 38
- Installation (JDK) 410
 - Klassenpfad anpassen (CLASSPATH) 416
 - Linux 412
 - Systempfad erweitern (PATH) 412
 - Windows 410
- instanceof 381, 428
- Instanzen 53
 - Instanzbildung 55
 - Schlüsselwort new 55
 - Zugriff 57
- Instanzbildung 55
- Instanzvariablen 120
- Interfaces *siehe* Schnittstellen

- Internet
 - URLs 347
- Interpreter 6
 - java 19
 - versus Compiler 6
- interrupt() 326
- invalidate() 224
- isDirectory() 149
- isFile() 149
- isInterrupted() 326
- ItemEvent (Klasse) 279
- ItemListener 214, 278
- itemStateChanged() 279
- Iteratoren 180

J

- Java
 - Anwendungen an Freunde weitergeben 9, 384
 - AWT 195 *siehe* AWT
 - Bytecode 7
 - Compiler 18
 - Dokumentation 417
 - Editoren 10
 - Entwicklungsumgebungen 10, 14
 - Entwicklungswerkzeuge 9
 - Geschichte 1
 - Groß- und Kleinschreibung 35
 - GUI-Anwendungen 197
 - Interpreter 19
 - JavaScript 3
 - JDK 9
 - JRE 384, 411
 - Klassenbibliothek 455
 - Pakete 60, 455
 - Plattformunabhängigkeit 8
 - Programme debuggen 381
 - Schlüsselwörter 419
 - Speicherbereinigung 123
 - Swing 195 *siehe* Swing
 - Syntax-Referenz 421
 - versus C# 2
 - versus C++ 2
- java (Interpreter) 19
- Java2D 245
- javac (Compiler) 18
- JavaDB 361
- JavaFX 350

JavaScript 3
 javaw 387
 JButton (Klasse) 200, 300
 JCheckBox (Klasse) 305
 JComboBox (Klasse) 276, 308
 JDBC 360
 – Treiber 360
 JDialog (Klasse) 282
 JDK 9
 – Deinstallation 412
 – Installation 410
 – Linux 412
 – Windows 410
 – Klassenpfad anpassen (CLASSPATH) 416
 – Systempfad erweitern (PATH) 412
 JFileChooser (Klasse) 259
 JFrame (Klasse) 199
 JLabel (Klasse) 298
 JList (Klasse) 308
 JMenu (Klasse) 313
 JMenuBar (Klasse) 257, 313
 JMenuItem (Klasse) 257, 313
 JOptionPane (Klasse) 342
 JPasswordField (Klasse) 305
 JRadioButton (Klasse) 235, 305
 JRE 384, 411
 JScrollBar (Klasse) 311
 JScrollPane (Klasse) 270, 311
 JTextArea (Klasse) 268, 302
 JTextComponent (Klasse) 268
 JTextField (Klasse) 268, 302
 JTextPane (Klasse) 269, 302

K

Keller 194
 KeyListener 213
 Klassen
 – abgeleitete Klasse 103
 – abstrakte 111
 – Arrays 59
 – Basisklasse 103
 – .class-Datei 75
 – Daten schützen 128
 – deklarieren 46
 – eines Programms 75
 – Felder 47
 – finalize() 123
 – Hauptklasse 75

 – innere Klassen 129
 – Instanzen 53
 – Instanzen auflösen 123
 – Instanzen bilden 55
 – Instanzvariablen 120
 – Klassentyp feststellen 428
 – Klassenvariablen 122
 – Konstruktoren 52
 – Mehrfachvererbung 130
 – Methoden 48, 114
 – Object 130
 – Pakete 60, 76
 – programmieren mit 53
 – Schlüsselwort abstract 111
 – Schlüsselwort class 46
 – Schlüsselwort final 110
 – Schlüsselwort new 55
 – Schlüsselwort public 129
 – Schnittstellen 132
 – Speicherbereinigung 123
 – statische Methoden 123
 – this 126
 – Übersicht 455
 – Vererbung 103
 – versus Schnittstellen 132
 – vor Instanzierung schützen 111
 – vor Vererbung schützen 110
 – wiederverwenden 74
 – Wrapper-Klassen 182
 – Zugriffsmodifizierer 78, 126, 129
 Klassenpfad 416
 Klassenvariablen 122
 Kombinationsfelder 308
 Kommandozeilenparameter 162
 Kommentare 21
 Komponenten
 – anordnen 201
 – Basisklasse Component 296
 – Befehlsnamen 232
 – Bildlaufleisten (JScrollbar) 311
 – Bildlaufleisten (JScrollPane) 270, 311
 – Container 200
 – Layout-Manager 202
 – Panel 232
 – setLayout() 202
 – Eingabefelder (JPasswordField) 305
 – Eingabefelder (JTextField und JTextArea) 302
 – Ereignisbehandlung 208

- getGraphics() 243
- Größe festlegen 227
- Größe festsetzen 401
- Hierarchie 296
- Hintergrundfarbe 227
- in Container aufnehmen 200
- in Fenster aufnehmen 200
- Kombinationsfelder (JComboBox) 276, 308
- Kurzinfos 319
- Layout-Manager 201
- Leinwand, an Fenstergröße anpassen 258
- Listenfelder (JList) 308
- Menüleisten 257, 313
- mit Symbolen 317
- Optionsfelder 235, 305
- Schaltflächen (JButton) 200, 300
- setBounds() 203
- statische Textfelder (JLabel) 298
- Titel ändern 399
- Vordergrundfarbe 227
- Konsole (Eingabeaufforderung) 12
- Konstanten 36
 - final 37
 - Literale 36
- Konstruktoren 52
 - Basisklassenkonstruktor aufrufen 109
 - in GUI-Anwendungen 200
 - Standardkonstruktor 136
- Kontrollkästchen 305
- Kontrollstrukturen 79
 - Bedingungen 79
 - Boolesche Ausdrücke 80
 - Mehrfachbedingungen 83
 - Schlüsselwort break 84
 - Schlüsselwort else 79
 - Schlüsselwort if 79
 - Schlüsselwort switch 83
 - Vergleichsoperatoren 81
 - Schleifen 85
 - Abbruchbedingungen 86
 - Endlosschleifen 86
 - for-Schleife 85, 88
 - Iteration abbrechen 91
 - Schleifenvariablen 85
 - Schlüsselwort break 91
 - Schlüsselwort continue 91
 - Schlüsselwort for 85, 88
 - Schlüsselwort while 85

- vorzeitig abbrechen 91
- while-Schleife 85
- Koordinatentransformationen 227
- Kurzinfos 319

L

- Lambda-Ausdrücke 375
- launch4j 387
- Layout-Manager 201
 - BorderLayout 399
 - FlowLayout 201
 - GridBagLayout 318
 - GridLayout 233
 - null-Layout 203
 - ohne Layout-Manager positionieren 203
 - Panel-Container 232
 - und Container 202
 - verschachtelte Container 232
- LinkedList (Klasse) 181
- Linux
 - Beispiele 467
 - CLASSPATH-Umgebungsvariable setzen 416
 - JDK-Installation 412
 - PATH-Umgebungsvariable setzen 412
- Liste 181
- Listenfelder 308
- listFiles() 149
- Literale 36
- Literatur 463
- LocalDate (Klasse) 174
- LocalDateTime (Klasse) 174
- LocalTime (Klasse) 174
- Look&Feel 214

M

- main() 24, 199
- Media (Klasse) 350
- MediaPlayer (Klasse) 350
- MediaTracker (Klasse) 264
- Menge 185
- Menüs 257, 313
 - aufbauen 257, 313
 - Ereignisbehandlung 257
 - in Fenster aufnehmen 313
 - Menübefehle 314
 - Menüleisten 257, 313
 - setJMenuBar() 313

- setMenuBar() 313
- Tastaturkürzel 257, 314
- MessageFormat (Klasse) 291
- Metal-Look 214
- Methoden 114
 - Basisklassenversion aufrufen 109
 - Datenaustausch 50
 - Deklaration 49, 114
 - lokale Variablen 50
 - main() 24, 199
 - ohne Methodenkörper 111
 - Parameter 115
 - Parameterübergabe 115
 - Arrays 118
 - call by reference 116
 - call by value 116
 - Polymorphie 108
 - Rückgabewert 115
 - Schlüsselwort abstract 111
 - Schlüsselwort final 110
 - Schlüsselwort return 115
 - Schlüsselwort static 123
 - Schlüsselwort void 51
 - Signatur 114
 - überschreiben 107f.
 - Überschreibung erzwingen 111
 - vor Überschreibung schützen 110
- MIDI 351
- MidiChannel (Schnittstelle) 353
- Modularisierung 66
 - durch Klassen 71
 - durch Methoden 70
- MouseAdapter (Klasse) 237
- mouseClicked() 400
- mouseDragged() 241
- MouseEvent (Klasse) 238
- MouseListener 213, 237
- MouseMotionAdapter (Klasse) 214, 242
- MouseMotionListener 214, 241
- mousePressed() 238

N

- Namensgebung
 - Variablen 36
- nativer Look 214, 218
- new 55
- next() 153
- nextInt() 153

- nextLine() 153
- Notepad 10
- Notepad++ 10
- now() 174

O

- Oberflächenelemente *siehe* Komponenten
- Object (Klasse) 130, 179
- Objektvariablen 116 *siehe* Instanzen
- of..() (LocalDateTime) 174
- OOP (Objektorientierte Programmierung) 45
 - Fehlerbehandlung mit Exceptions 92
 - Kapselung 81
 - Klassen 46
 - Objekte 46
 - Polymorphie 108
 - Vererbung 103
- Operatoren 37
 - Assoziativität 429
 - Auswertungsreihenfolge 429
 - Bitoperatoren 427
 - Division 43
 - Inkrement 38
 - Klammern 429
 - Modulo 100
 - Operandenauswertung 430
 - Priorität 40, 429
- Optionsfelder 305

P

- pack() 201
- package 76
- paint() 222
- paintComponent() 222f., 258
- Pakete 60
 - anlegen 76
 - importieren 60
 - Schlüsselwort import 60, 77
 - Schlüsselwort package 76
 - und Zugriffsregelung 129
- Panels
 - Fenster-Layout 232
- Parameter
 - bei der Ereignisbehandlung 208
 - final 116
 - Übergabe an Anwendungen 162
 - von Methoden 115

parse() (LocalDateTime) 174
 PATH-Umgebungsvariable 412
 Period (Klasse) 177
 Polymorphie 108
 print() 140, 290
 printf() 142
 println() 51, 140 f.
 printStackTrace() 96
 PrintStream (Klasse) 141
 PrintWriter (Klasse) 145, 147
 process() 340
 Programmerstellung 5

- Ablauf 5
- Fehlersuche 381

 Programmierbeispiele

- Bildbetrachter 253
- BilderAnimation 340
- Dateiliste 149
- drucken 290
- Funktionenplotter 225, 229
- Hashtabelle 189
- Instanzenzähler 120
- JavaDBDemo 365
- Kaninchen 89
- Keller 194
- KlavierDemo 354
- Listen 183
- Sichelzellenanämie 164
- Stack 194
- Swing-Look&Feel 216
- Texteditor 267
 - Dateien laden 273, 275
 - Entwurf 267
 - Font 272
 - JTextArea 268
 - Menüleiste 267
 - nach Textstellen suchen 284
 - Textsuche fortsetzen 405
 - Zwischenablage 287
- Zeitanzeige 327, 334
- Zinsrechnung 66

 Prompt 12
 Prozess 321
 public 78, 129
 publish() 340

R

Random (Klasse) 172
 read() (FileReader) 155
 readLine() 149
 Referenzen 55
 renameTo() 149
 repaint() 223
 ResultSet (Schnittstelle) 363
 resume() 326
 return 115
 runFinalization() 124

S

Scanner (Klasse) 151
 Schaltflächen 300

- mit Symbolen 317

 Schleifen 85
 Schlüsselwörter 419

- abstract 111
- break 84, 91
- case 83
- catch 95
- class 46
- continue 91
- else 79
- extends 106
- final 37, 110, 116
- finally 97, 156
- for 85, 88
- if 79
- implements 133
- import 60, 77
- new 55
- package 76
- return 115
- static 122 f.
- super 109
- switch 83
- synchronized 332
- this 126
- throw 94
- try 95, 157
- void 51
- while 85

 Schnittstellen 132

- Ableitung 133
- ActionListener 208, 257, 282, 288

- addActionListener() 209
- addItemListener() 278
- addMouseListener() 400
- addMouseMotionListener() 404
- Comparable 191
- Deklaration 132
- FocusListener 213
- ItemListener 214, 278
- KeyListener 213
- Listener-Schnittstellen 207
- MouseListener 213
- MouseMotionListener 214, 241
- Nutzen 132
- Runnable 329
- Schlüsselwort implements 133
- TextListener 214
- und Ereignisbehandlung 207
- versus Klassen 132
- WindowListener 210, 214
- Schriftarten *siehe* Fonts
- Sequencer (Schnittstelle) 351, 352
- Servlets 3
- setAccelerator() 257
- setBounds() 203
- setDefaultCloseOperation() 212
- Set (Klasse) 185
- setLayout() 201f.
- setLookAndFeel() 215
- setToolTipText() 319
- setVisible() 200
- showConfirmDialog() 286
- showInputDialog() 287
- showMessageDialog() 287
- showOpenDialog() 259
- Sichelzellenanämie 164
- sleep() 326
- Sortieren (Arrays) 191
- Sortieren (Listen) 192
- SQL 359, 363
 - ORDER BY-Klausel 407
 - SELECT 363
 - WHERE-Klausel 407
- Stack 194
- Stack (Klasse) 194
- Standardausgabe 140
- Standardeingabe 140
- start() (Thread) 326
- Statement (Schnittstelle) 363
- static 122f., 439
- Steuerelemente *siehe* Komponenten
- Stilkonventionen 453
- stop() (Thread) 326
- Streams 139
 - Standardausgabe 140
 - Standardeingabe 140
 - System.in 140
 - System.out 140
- Strings 158
 - aneinanderhängen 158
 - Klasse String 158
 - Klasse StringBuffer 163
 - Klasse StringBuilder 155, 163
 - Klasse StringTokenizer 177
 - konkatenieren 158
 - konvertieren 159
 - Länge 159
 - nach Teilstrings suchen 286, 405
 - Speicherverwaltung 163
 - StringBuilder-Methoden 165
 - String-Methoden 161
 - vergleichen 160
 - zerlegen 177
- String (Klasse) 158
- StringBuffer (Klasse) 163
- StringBuilder (Klasse) 155, 163
- StringTokenizer (Klasse) 177
- Suchen (Arrays) 192
- Suchen (Listen) 192
- super 109, 381
- suspend() 326
- Swing 195
 - Ereignisbehandlung für Fenster 210f.
 - Klassen
 - AbstractButton 306
 - ButtonGroup 236, 306
 - DefaultListModel 310
 - Document 275
 - ImageIcon 261
 - ImageIO 263
 - JButton 200, 300
 - JCheckBox 305
 - JComboBox 276, 308
 - JDialog 282
 - JFileChooser 259
 - JFrame 199
 - JLabel 298
 - JList 308
 - JMenu 257, 313

- JMenuBar 257, 313
- JMenuItem 257, 313
- JOptionPane 286
- JPanel 233
- JPasswordField 305
- JRadioButton 235, 305
- JScrollBar 311
- JScrollPane 270, 311
- JTextArea 268, 302
- JTextComponent 268
- JTextField 268, 302
- JTextPane 269, 302
- Look&Feel 214
- Metal-Look 214
- Metal-Themes 214
- native Look 214, 218
- UIManager 214
- Windows-Look 218
- SwingUtilities 333
- SwingWorker (Klasse) 340
- switch 83
- Synchronisierung 331
- synchronized 332
- Syntaxreferenz 421
- System (Klasse) 124
- System.in 140
- System.out 140
- System-Piep 208

T

Tabellen (Datenbanken) 358

Tastaturkürzel 257

Text 267

- drucken 290
- Editierbarkeit festlegen 293
- EditorKits 288
- Einfügemarke setzen 273
- Font 272
- JTextArea 268
- JTextField 268
- JTextPane 269
- nach Textstellen suchen 284
- Textdateien laden (und speichern) 273
- Textsuche fortsetzen 405
- verfügbare Fonts ermitteln 293
- Zwischenablage 287

TextArea (Klasse) 302

Textfelder 302

- statische 298

TextListener 214

this 126

Threads 321

- Animationen 334
- Ausführung starten 326
- currentThread() 326
- GUI-Manipulation 333
- interrupt() 326
- isInterrupted() 326
- Klasse Thread 325
- präemptives Multithreading 322
- Prioritäten 331
- Prozesse 321
- resume() 326
- Runnable-Schnittstelle 329
- setPriority() 331
- sleep() 326
- start() 326
- stop() 326
- suspend() 326
- Swing 340
- Synchronisierung 331

Thread (Klasse) 325

thread-safe 334

throw 94

Token 152

Toolkit (Klasse) 209, 264

ToolTips 319

Treiber (Datenbanken) 360

try 95, 157

Typidentifizierung 428

Typumwandlung 40

- Autoboxing 182
- automatische 41
- Casting 42
- Division 43
- explizite 42
- Strings 159

U

Umlaute 141, 147

Unicode 34

updateComponentTreeUI() 216

URL (Klasse) 348

URL (Uniform Resource Locator) 347

V

Variablen

- Arbeit mit 32
- Bereichsüberschreitung 42
- Deklaration 30
- Felder 47
- final 37
- Initialisierung 32
- Instanzen 53
- Instanzvariablen 120
- Kategorien 120
- Klassenvariablen 122
- lokale 50
- Namensgebung 36
- Parameter 115
- Schleifenvariablen 85
- Schlüsselwort static 122
- schützen 128
- Typumwandlung 40
- Verdeckung 125
- Wert 31

Vererbung 103

- abgeleitete Klasse 103
- abgeleitete Objekte als Basisklassenobjekte 108, 113
- abstrakte Klassen 111
- Basisklasse 103
- Basisklasse festlegen 106
- Basisklassenkonstruktor aufrufen 109
- final-Klassen 110
- geerbte Methoden überschreiben 107f.
- Object 130
- Polymorphie 108
- private Elemente 135
- Schlüsselwort abstract 111
- Schlüsselwort extends 106

- Schlüsselwort final 110
 - Schlüsselwort super 109
 - Schnittstellen 132
 - Standardkonstruktor 135
 - überschriebene Methoden aufrufen 109
- void 51

W

Wachstumsprozesse 89

while 85

Windows

- CLASSPATH-Umgebungsvariable setzen 416
 - JDK-Installation 410
 - PATH-Umgebungsvariable setzen 412
- WindowAdapter (Klasse) 211
- windowClosing() 211
- WindowListener 210, 214
- Windows-Look 218
- Wrapper-Klassen 182
- write() (FileWriter) 148

Z

Zahlensysteme 28

Zeichenfolgen *siehe* StringsZeichnen *siehe* Grafik

Zeitmessung 176

Zeit- und Datumsangaben

- erzeugen 174
- Formate 175
- rechnen mit 175

Zeit- und Datumsanzeige 173

Zufallszahlen 171

Zugriffsmodifizierer 78, 129

Zwischenablage 287