

Schleifen schreiben, statt sich im Kreis zu drehen
6., überarbeitete und aktualisierte Auflage

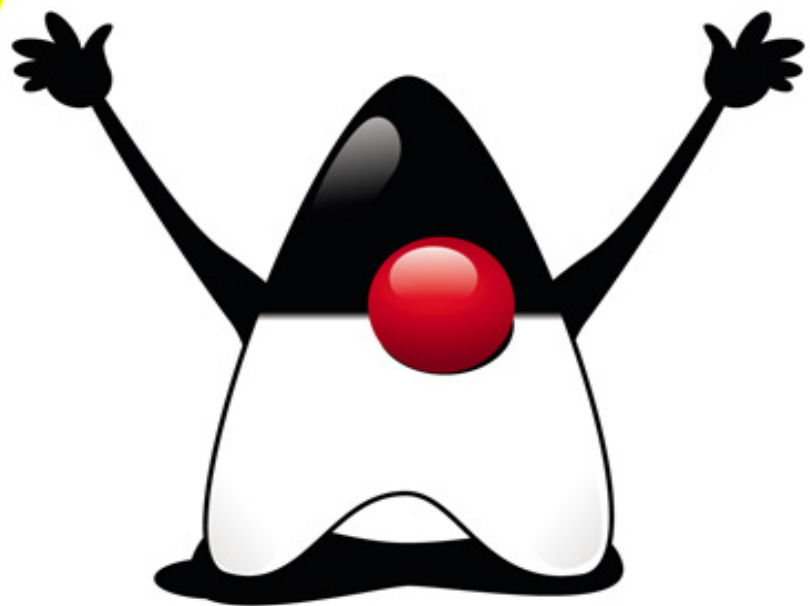
Java[®]

FÜR

DUMMIES[®]

Auf einen Blick:

- Die Grundlagen eines Java[®]-Programms kennenlernen
- Objektorientierte Programme erstellen
- Fortgeschrittene Java-Techniken lernen
- Java-Applets schreiben



Barry Burd

Nennen Sie es Java-»Syntax« oder Java-»Grammatik« – wenn Sie Java-Code schreiben, müssen Sie wissen, wohin die Schlüsselwörter, die Semikolons und all die anderen Dinge gehören. Diese Schummelseite fasst die Regeln der Java-Syntax zusammen und liefert zu jedem Befehl ein paar Informationen.

Java-Befehle

Die meisten Zusammenfassungen der Syntax listen die Einzelheiten aller Befehle auf und verwenden dabei Beschreibungen, die formell sind und einer Geheimsprache ähneln. Die hier in den Beispielen gezeigten Befehlstypen enthalten eine kurze informative Beschreibung dessen, was der Befehlstyp macht. Die Beschreibung der Java-Syntax ist weit davon entfernt, umfassend zu sein, aber Sie werden sehen, dass sie lesbar und nützlich ist.

Eine Java-Klasse:

```
class PreIncrementDemo {  
  
}
```

Die Daten innerhalb der geschweiften Klammern gehören zu einer Klasse mit dem Namen `PreIncrementDemo`. Diese Klasse ist eine Art Blaupause für das Erstellen von Objekten (und Instanzen der Klasse `PreIncrementDemo`).

Importdeklarationen:

```
import java.util.Scanner;  
  
import static java.lang.System.out;
```

Wenn irgendwo in dieser `.java`-Datei der Name `Scanner` verwendet wird, dient er als Abkürzung für `java.util.Scanner`. Dies bedeutet, dass der Name `Scanner` die Klasse `Scanner` repräsentiert, die Bestandteil des Pakets `java.util` ist.

Wenn irgendwo in der Java-Datei der Name `out` verwendet wird, dient er als Abkürzung für `java.lang.System.out`. Dies bedeutet, dass der Name `out` genommen wird, um das statische Feld `out` darzustellen, das Bestandteil der Klasse `System` ist, die sich wiederum im Paket `java.lang` befindet.

Die Deklaration von Variablen:

```
int numberOfBunnies = 27;  
  
JFrame meinFrame;
```

Der Name `numberOfBunnies` speichert einen `int`-Wert (eine Ganzzahl wie 42, 0, -284) und der Anfangswert von `numberOfBunnies` ist 27. Der Name `meinFrame` verweist auf ein Objekt, das aus der Klasse `JFrame` erstellt worden ist. (Das bedeutet, dass `meinFrame` auf eine *Instanz* der Klasse `JFrame` verweist. Die Klasse `JFrame` liegt fertig definiert in einem Paket mit dem Namen `javax.swing` vor.)

Zuweisungsbefehle:

```
standDesKontos = 50.22;  
standDesKontos *= 1.05;  
standDesKontos = standDesKontos + 1000000.00;
```

Der Variablen `standDesKontos` wird der Wert 50.22 zugewiesen. Dann wird der Wert von `standDesKontos` mit 1.05 multipliziert. Zum Schluss wird dem Wert von `standDesKontos` 1000000.00 hinzugefügt.

Die Deklaration einer Methode:

```
int incrementBy(int amount) {  
  
}
```

Es wird eine Methode mit dem Namen `incrementBy` eingeführt (*deklariert*). Diese Methode nimmt als Eingabe (als *Parameter*) einen `int`-Wert. Im Körper der Methode (das ist der Bereich zwischen den geschweiften Klammern) kennt der Variablenname `amount` den Wert des Parameters. Wenn Sie die Methode `incrementBy` ausführen, wird (möglicherweise) ein (anderer) Wert zurückgegeben, der aber auch vom Typ `int` ist.

Aufrufe von Methoden:

```
n = incrementBy(6);  
  
System.out.println(incrementBy(10));
```

Das Aufrufen der Methode `incrementBy` (das heißt das Ausführen der Anweisungen im Körper der Methode `incrementBy`) legt als Wert des Parameters der Methode 6 fest. Der Wert, den die Methode aufgrund des Aufrufs zurückgibt, wird genommen und der Variablen `n` zugewiesen.

Außerdem wird die Methode `incrementBy` mit dem Parameterwert 10 aufgerufen. Dann wird der Wert, der durch diesen Methodenaufruf zurückgegeben wird, genommen und an eine andere Methode übergeben – hier an die vordefinierte Java-Methode `System.out.println`.

Eine Methode »main«:

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

Ein Java-Programm beginnt seine Aktivitäten damit, dass die Befehle, die sich im Körper der Methode mit dem Namen `main` befinden, ausgeführt werden.

Der Aufruf eines Konstruktors:

```
JFrame myFrame = new JFrame("Dies ist mein Frame!");
```

Es wird ein neues Objekt erstellt und dabei die Definition der Klasse `JFrame` verwendet. (Ein `JFrame` ist etwas, das Sie in Programmen normalerweise als *Fenster* bezeichnen.) In der Titelzeile des neuen Objekts werden die Wörter `Dies ist mein Frame!` angezeigt. Die Variable `myFrame` sollte auf das neue Objekt (die neue Instanz der Klasse `JFrame`) verweisen.

Ein »if«-Befehl:

```
if (eingegebeneZahl == zufallsZahl) {  
  
} else {  
  
}
```

Wenn (*if*) eingegebeneZahl gleich zufallsZahl ist, sollen alle Anweisungen ausgeführt werden (die sich innerhalb der ersten geschweiften Klammern befinden). Wenn eingegebeneZahl aber nicht gleich der zufallsZahl ist, sollen andere Befehle ausgeführt werden (die Befehle, die sich im zweiten Paar geschweiften Klammern – hinter dem Worte *else* (deutsch *anderenfalls*) – befinden).

Eine »switch«-Anweisung:

```
switch (strophe) {  
  
case 1:  
  
    break;  
  
case 2:  
  
    break;  
  
case 3:  
  
    break;  
  
default:  
  
    break;  
  
}
```

Wenn strophe den Wert 1 hat, sollen die Befehle hinter case 1 ausgeführt werden. Danach bricht das Programm aus dem normalen Ablauf aus (*break*) und verlässt den gesamten switch-Befehl. Wenn strophe den Wert 2 hat, sollen die Befehle hinter case 2 ausgeführt werden. Danach bricht das Programm aus dem normalen Ablauf aus und verlässt den gesamten switch-Befehl. Bla, bla, bla für strophe gleich 3. Wenn der Wert von strophe aus irgendwelchen Gründen weder 1 noch 2 oder 3 ist, werden die Befehle ausgeführt, die hinter default stehen. Danach wird die gesamte switch-Anweisung verlassen.

Ein »while«-Befehl:

```
while (eingegebeneZahl != zufallsZahl) {  
  
}
```

Es wird geprüft, ob die `eingegebeneZahl` ungleich der `zufallsZahl` ist. Ist dies der Fall (wenn also die beiden Zahlen nicht gleich sind), werden die Befehle zwischen den geschweiften Klammern ausgeführt, und der gesamte Ablauf wird wiederholt. (Es wird überprüft, ob die Zahlen ungleich sind, und so weiter.) Wenn es sich während einer der Wiederholungen herausstellt, dass die Bedingung `eingegebeneZahl != zufallsZahl` den Wert `true` zurückgibt (eingegebeneZahl also gleich `zufallsZahl` ist), wird das Wiederholen der Anweisungen beendet und mit den Befehlen weitergemacht, die unmittelbar auf den `while`-Befehl folgen.

Ein »for«-Befehl:

```
for (int count = 1; count <= 10; count++) {  
  
}
```

Beim Zählen (englisch *to count*) sollte mit 1 begonnen werden. In obigem Beispiel muss geprüft werden, ob `count` kleiner oder gleich 10 ist. Da `count` kleiner als 10 ist, werden die Befehle in den geschweiften Klammern ausgeführt. Wenn dies abgeschlossen worden ist, wird `count++` ausgeführt (der Wert von `count` wird um 1 erhöht).

Es wird nun wieder geprüft, ob `count` immer noch kleiner oder gleich 10 ist. (In diesem Beispiel hat `count` nun den Wert 2.) Da `count` kleiner als 10 ist, werden wieder die Befehle in den geschweiften Klammern ausgeführt und der Wert von `count` wieder um 1 erhöht.

Wenn beim Start einer Wiederholung herausgefunden wird, dass `count <= 10` nicht mehr wahr ist, werden die Wiederholungen sofort beendet und die Befehle ausgeführt, die unmittelbar auf den `for`-Befehl folgen.

Ein »do ... while«-Befehl:

```
do {  
  
} while (antwort != 'j' && antwort != 'n');
```

Am Anfang wird nichts geprüft, sondern sofort mit der Ausführung der Bedingungen begonnen, die sich zwischen den geschweiften Klammern befinden. Danach wird geprüft, ob `antwort` ungleich 'j' und ungleich 'n' ist (`antwort` darf also weder 'j' noch 'n' sein). Wenn diese Bedingung erfüllt ist, wird alles wiederholt. (Die Befehle zwischen den geschweiften Klammern werden ausgeführt, und dann wird wieder geprüft, ob `antwort` 'j' oder 'n' ist.) Wenn am Ende einer Wiederholung herausgefunden wird, dass `antwort` 'j' oder 'n' ist, werden die Wiederholungen abgebrochen und die Befehle ausgeführt, die unmittelbar auf den `do ... while`-Befehl folgen.

Eine »enum«-Deklaration:

```
enum Zimmer {ballsaal, kueche, esszimmer,  
             wohnzimmer, empfangshalle, arbeitszimmer,  
             bibliothek, billardzimmer, wintergarten};  
  
Zimmer sceneOfTheCrime = Zimmer.arbeitszimmer;
```

Jede Variable, deren Deklaration vom Typ Zimmer ist, speichert einen dieser Werte: Zimmer.ballsaal, Zimmer.kueche, Zimmer.esszimmer, Zimmer.wohnzimmer und so weiter. In diesem Beispiel wird die Variable sceneOfTheCrime so deklariert, dass sie vom Typ Zimmer ist, und sie beginnt den Ablauf damit, dass sie Zimmer.arbeitszimmer speichert.

Ein erweiterter »for«-Befehl:

```
for (Zimmer meinZimmer : Zimmer.values()) {  
  
}
```

Es werden die Befehle in den geschweiften Klammern mehrmals ausgeführt – einmal für jeden der enum-Werte von Zimmer. Die Befehle werden einmal mit Zimmer ist gleich Zimmer.ballsaal ausgeführt. Dann werden die Befehle noch einmal ausgeführt, aber dieses Mal mit Zimmer ist gleich Zimmer.kueche. Und so weiter.

Eine Klasse erweitern:

```
class VollzeitMitarbeiter extends Mitarbeiter {  
  
}
```

Ein VollzeitMitarbeiter ist ein Mitarbeiter mit zusätzlichen Funktionen. Hier geht es insbesondere um Funktionen, die zwischen den geschweiften Klammern beschrieben werden.

In diesem Beispiel ist VollzeitMitarbeiter eine *Unterklasse* (oder *Kindklasse*) der Klasse Mitarbeiter, die gleichzeitig die *Oberklasse* (oder *Elternklasse*) der Klasse VollzeitMitarbeiter ist.

Eine »throws«-Klausel:

```
void doStuff() throws IOException {
```

```
    Scanner diskScanner =  
        new Scanner(new File("MitarbeiterInfo.txt"));
```

Wenn Sie Code in der Methode doStuff ausführen, könnte dies eine IOException werfen (englisch *to throw*).

Wenn Sie versuchen, eine neue Instanz der Klasse `File` zu erstellen, kann einiges schiefgehen. Java besteht darauf, dass Sie mögliche Unannehmlichkeiten bestätigen, indem Sie ankündigen, dass Ihre Methode in der Lage ist, eine `IOException` zu werfen. Wenn Sie Code ausführen, der eine `IOException` werfen kann, müssen Sie diese Ausnahme (das bedeutet *Exception* auf Deutsch) in Ihrem Code bestätigen. Eine Ausnahme, die bestätigt werden muss, wird *geprüfte Ausnahme* genannt.

Konstrukturen:

```
class Temperature {  
  
    private double number;  
  
    private ScaleName scale;  
  
    public Temperature() {  
        number = 0.0;  
        scale = ScaleName.fahrenheit;  
    }  
  
    public Temperature(double number, ScaleName scale) {  
        this.number = number;  
        this.scale = scale;  
    }  
}
```

Sie können ein Objekt `Temperature` (eine Instanz der Klasse `Temperature`) erstellen, indem Sie `new Temperature()` (mit leeren Klammern) aufrufen. Sie können ein neues Objekt `Temperature` auch über den Aufruf von `new Temperature(70.0, ScaleName.fahrenheit)` oder `new Temperature(0.0, ScaleName.celsius)` oder mit einem anderen Namen für eine Temperaturskala anlegen.

Die Deklaration eines Konstruktors sieht wie die Deklaration einer Methode aus, wobei aber der Name des Konstruktors mit dem Namen der entsprechenden Klasse übereinstimmt. Darüber hinaus verfügt die Deklaration eines Konstruktors nicht über einen Rückgabebetyp (noch nicht einmal über `void`).

Den Konstruktor einer Oberklasse aufrufen:

```
class TemperatureNice extends Temperature {  
  
    public TemperatureNice() {  
        super();  
    }  
}
```

```
public TemperatureNice(double number) {
    super(number);
}
```

Wenn das Programm `new TemperatureNice()` erweitert (indem der parameterlose Konstruktor `TemperatureNice` aufgerufen wird), soll das Programm `super()` aufrufen. In diesem Beispiel bedeutet der Aufruf von `new Temperature()`, dass der parameterlose Konstruktor `Temperature` aufgerufen wird.

Wenn das Programm `new TemperatureNice(23.0)` erweitert wird (indem der mit einem Parameter versehene Konstruktor `TemperatureNice` aufgerufen wird), soll das Programm `super(23.0)` aufrufen. In diesem Beispiel bedeutet der Aufruf von `new Temperature(23.0)`, dass der einen Parameter aufweisende Konstruktor `Temperature` aufgerufen wird.

Mit Arrays arbeiten (Teil 1):

```
int gaeste[] = new int[10];
```

Es wird eine Variable mit dem Namen `gaeste` erstellt. Diese Variable verweist auf ein Array aus `int`-Werten. Dieses Array hat Platz für zehn `int`-Werte, und `gaeste` kann auf diese Werte verweisen.

Mit Arrays arbeiten (Teil 2):

```
for (int zimmerNum = 0; zimmerNum < 10; zimmerNum++) {
    gaeste[zimmerNum] = 0;
}
```

In jedem Zimmer des Arrays wird für `gaeste` der Wert `0` eingetragen.

Mit Arrays arbeiten (Teil 3):

```
int belegung[] = {4, 4, 2, 4, 2, 2, 4, 2, 0, 2};
```

Es wird eine Variable mit dem Namen `belegung` angelegt, die auf ein Array aus `int`-Werten verweist. Dieses Array speichert die `int`-Werte `4, 4, 2` und so weiter. Die Variable `belegung` verweist auf diesen Speicherort.

Mit Arrays arbeiten (Teil 4):

```
Zimmer zimmer[] = new Zimmer[10];
```

Es wird eine Variable mit dem Namen `zimmer` angelegt, die auf ein `Zimmer`-Array verweist. (`Zimmer` und `zimmer` sind aufgrund der Schreibweise unterschiedliche Objekte!) Die Variable `zimmer` verweist auf das Array.

Zugriffsmodifizierer verwenden (Teil 1):

```
package com.allmycode.utilities;
```

```
public class First {  
  
}
```

```
class Second {  
  
}
```

Der Code dieses Beispiels gehört zu einem Paket mit dem Namen `com.allmycode.utilities`. Die Klasse `First` steht jedem Java-Code zur Verfügung. Die Klasse `Second` steht Java-Code nur dann zur Verfügung, wenn er sich ebenfalls im Paket `com.allmycode.utilities` befindet.

Zugriffsmodifizierer verwenden (Teil 2):

```
package com.allmycode.utilities;
```

```
class Third {  
  
    public amountInAccount = 50.22;  
  
    int numOfThings = 22;  
  
    private int stuff = 10;  
  
    protected int x = 40;
```

Der Code hier gehört zu einem Paket mit dem Namen `com.allmycode.utilities`. Jeder Java-Code kann auf die Variable `amountInAccount` zugreifen. (Jeder Java-Code kann auf `amountInAccount` verweisen.) Die Variable `numOfThings` steht Java-Code nur dann zur Verfügung, wenn sich dieser Code ebenfalls im Paket `com.allmycode.utilities` befindet. Auf die Variable `stuff` kann von außerhalb der Klasse `Third` nicht zugegriffen werden. Die Variable `x` steht nur für Java-Code zur Verfügung, wenn sich dieser im Paket `com.allmycode.utilities` befindet oder Bestandteil einer Unterklasse der Klasse `Third` ist.

Ein Interface implementieren:

```
class GameFrame extends JFrame  
  
    implements ActionListener, ItemListener {  
  
  
}
```

Für körperlose Methoden, die in ActionListener- und ItemListener-Interfaces deklariert werden, werden Körper (Implementierungen) zur Verfügung gestellt.

Ein Lambda-Ausdruck:

```
(kunde) -> kunde.name.equals("Burd")
```

Gibt true zurück, wenn der Name des Kunden Burd ist.

```
kundenListe.stream()
```

```
.filter((kunde) -> kunde.name.equals("Burd"))
```

Es werden die Objekte in kundenListe genommen und mit der Methode stream nacheinander an eine filter-Methode gesendet. Die filter-Methode wendet den Lambda-Ausdruck auf jedes Objekt an und erhält für jeden Kunden, dessen Name Burd ist, true zurück. Wenn der Name nicht mit Burd übereinstimmt, wird false zurückgeliefert. Die filter-Methode sendet dann alle Kunden mit dem Namen Burd an die Methode, die in der entsprechenden Verarbeitungszeile als nächste folgt.

Eine Methodenreferenz:

```
Kunde::isBurd
```

Es werden die Befehle ausgeführt, die sich im Körper der Methode isBurd der Klasse Kunde befinden.

```
class Kunde {
```

```
    String name;
```

```
    public boolean isBurd() {
```

```
        return name.equals("Burd");
```

```
    }
```

```
    // Hier kommt weiterer Code hin
```

```
}
```

```
kundenListe.stream().filter(Kunde::isBurd)
```

Es werden die Objekte in kundenListe genommen und mithilfe der Methode stream nacheinander an die filter-Methode gesendet. Diese Methode wendet auf jedes Objekt die Methode isBurd der Klasse Kunde an. Dabei wird für Kunden mit dem Namen Burd der Wert true zurückgeliefert, während bei allen anderen Kunden false zurückgegeben wird. Die filter-Methode sendet dann nur die Kunden mit dem Namen Burd an die Methode, die als nächste in der Verarbeitungszeile steht.

Barry Burd

Java für Dummies

*Übersetzung aus dem Amerikanischen
von Jutta Schmidt*

6., überarbeitete und aktualisierte Auflage

WILEY

WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

WILEY-VCH Verlag GmbH & Co. KGaA

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

6., überarbeitete und aktualisierte Auflage 2014

© 2014 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim

Original English language edition »Java For Dummies« © 2014 by John Wiley and Sons, Inc.

All rights reserved including the right of reproduction in whole or in part in any form. This translation published by arrangement with John Wiley and Sons, Inc. This EBook published under license with the original publisher John Wiley and Sons, Inc.

Copyright der englischsprachigen Originalausgabe »Java For Dummies« © 2014 by John Wiley and Sons, Inc.

Alle Rechte vorbehalten inklusive des Rechtes auf Reproduktion im Ganzen oder in Teilen und in jeglicher Form. Dieses E-Book wird mit Genehmigung des Original-Verlages John Wiley and Sons, Inc. publiziert.

Wiley, the Wiley logo, Für Dummies, the Dummies Man logo, and related trademarks and trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries. Used by permission.

Wiley, die Bezeichnung »Für Dummies«, das Dummies-Mann-Logo und darauf bezogene Gestaltungen sind Marken oder eingetragene Marken von John Wiley & Sons, Inc., USA, Deutschland und in anderen Ländern.

Das vorliegende Werk wurde sorgfältig erarbeitet. Dennoch übernehmen Autoren und Verlag für die Richtigkeit von Angaben,

Hinweisen und Ratschlägen sowie eventuelle Druckfehler keine Haftung.

Coverfoto: © Silveira Neto – <http://silveiraneto.net>

Korrektur: Petra Heubach-Erdmann und Jürgen Erdmann,
Düsseldorf

Satz: inmedialo Digital- und Printmedien UG, Plankstadt

Print ISBN: 978-3-527-71070-6

ePDF ISBN: 978-3-527-68673-5

ePub ISBN: 978-3-527-68675-9

mobi ISBN: 978-3-527-68674-2

Inhaltsverzeichnis

Über den Autor

Einführung

Wie Sie an dieses Buch herangehen sollten

Konventionen, die in diesem Buch verwendet werden

Was Sie nicht lesen müssen

Ein paar einfache Annahmen

Wie dieses Buch aufgebaut ist

Teil I: Los geht's

Teil II: Eigene Java-Programme schreiben

Teil III: Das große Ganze im Auge behalten: die objektorientierte Programmierung

Teil IV: Fortgeschrittene Java-Techniken

Teil V: Der Top-Ten-Teil

Symbole, die in diesem Buch verwendet werden

Anmerkung der Übersetzerin

Wie es weitergeht

Teil I

Los geht's

1 Alles über Java

Was Sie mit Java machen können

Warum Sie Java verwenden sollten

Einen Überblick erhalten: Wie sich Java einordnen lässt

Objektorientierte Programmierung (OOP)

Objektorientierte Sprachen

Objekte und ihre Klassen

Was ist das Besondere an einer objektorientierten Sprache?

Ihr Verständnis für Klassen und Objekte vertiefen

Wie geht es weiter?

2 Alles über Software

Anleitungen für einen Schnellstart

Was Sie auf Ihrem Computer installieren

Was ist ein Compiler?

Was ist eine Java Virtual Machine?

Software entwickeln

Was ist eine integrierte Entwicklungsumgebung?

3 Die grundlegenden Teile verwenden

Die Sprache Java sprechen

Die Grammatik und die gebräuchlichen Bezeichnungen

Die Wörter in einem Java-Programm

Sich zum ersten Mal mit Java-Code beschäftigen

Ein einfaches Java-Programm verstehen

Die Java-Klasse

Die Java-Methode

Die Methode »main« eines Programms

Wie Sie dem Computer letztendlich mitteilen, dass er etwas tun soll

Geschweifte Klammern

Kommentare, Kommentare

Dem Code Kommentare hinzufügen

Und wie sieht Barrys Ausrede aus?

Kommentare verwenden, um mit dem Code zu experimentieren

Teil II

Eigene Java-Programme schreiben

4 Das Optimum aus Variablen und ihren Werten herausholen

Eine Variable variieren

Zuweisungsbefehle

Wertetypen, die Variablen annehmen können

Text anzeigen

Ganzzahlen

Deklarationen kombinieren und Variablen initialisieren

Die Atome: Javas primitive Typen

Der Typ »char«

Der Typ »boolean«

Die Moleküle und Verbindungen: die Referenztypen

Eine Importdeklaration

Mit Operatoren neue Werte erstellen

Einmal initialisieren, mehrfach zuweisen

Inkrement- und Dekrement-Operatoren

Zuweisungsoperatoren

5 Den Programmablauf mit entscheidungsfindenden Befehlen steuern

Entscheidungen fällen (»if«-Befehle in Java)

Eine Zahl raten

Tastatureingaben kontrollieren

Zufallszahlen

Der Befehl »if«

Das doppelte Gleichheitszeichen

Nichts als Klammern

Den Code bei »if«-Befehlen einrücken

Elselos in Ifrika

Bedingungen mit Vergleichsoperatoren und mit logischen Operatoren bilden

Zahlen miteinander vergleichen: Vergleichsoperatoren

Objekte vergleichen

Alles auf einen Schlag importieren

Javas logische Operatoren

Vive les nuls!

(Bedingungen in Klammern)

Verschachtelungen

Unter mehreren Alternativen wählen (Javas »switch«-Befehle)

Der »switch«-Befehl

To break or not to break

Der neue, verbesserte »switch«-Befehl

6 Den Programmablauf mit Schleifen steuern

Anweisungen mehrfach wiederholen (der Java-Befehl »while«)

Eine bestimmte Anzahl an Wiederholungen festlegen (der Java-Befehl »for«)

Die Anatomie eines »for«-Befehls

Die Weltpremiere von »Al's All Wet«

Wiederholen, bis Ihr Wunsch erfüllt wird (der Java-Befehl »do«)

Ein einzelnes Zeichen lesen

Java und die Behandlung von Dateien

Variablendeklarationen und Blöcke

Teil III

Das große Ganze im Auge behalten: Die objektorientierte Programmierung

7 In Begriffen wie Klassen und Objekte denken

Eine Klasse definieren (was es heißt, ein Konto zu sein)

Variablen deklarieren und Objekte erstellen

Eine Variable initialisieren

Die Felder eines Objekts verwenden

Ein Programm; mehrere Klassen

Öffentliche (»public«) Klassen

Eine Methode in einer Klasse definieren (ein Konto anzeigen)

Ein Konto, das sich selbst anzeigt

Der Kopf der Methode »display«

Werte an Methoden senden und von dort erhalten (Zinsen berechnen)

Einen Wert an eine Methode übergeben

Einen Wert von der Methode »getZinsen« zurückgeben

Zahlen gut aussehen lassen

Einzelheiten mit Zugriffsmethoden verbergen

Gute Programmierung

Öffentliches Leben und private Träume: ein Feld sperren

Regeln mit Zugriffsmethoden erzwingen

8 Zeit und Geld sparen: Code wiederverwenden

Eine Klasse definieren (was es bedeutet, ein Mitarbeiter zu sein)

Das letzte Wort an Mitarbeiter

Die Klasse gut verwenden

Gehalt bezahlen

Mit Dateien auf der Festplatte arbeiten (ein kleiner Umweg)

Daten in einer Datei ablegen

Code kopieren und einfügen

Eine Datei auslesen

Wer hat die Datei verschoben?

Den Dateinamen um den Verzeichnisnamen erweitern

Eine Zeile nach der anderen lesen

Die Verbindung zu einer Datei auf der Festplatte beenden

Unterklassen definieren (was es bedeutet, ein Vollzeit- oder ein Teilzeitmitarbeiter zu sein)

Unterklassen erstellen

Das Anlegen von Unterklassen kann zur Gewohnheit werden

Mit Unterklassen arbeiten

Typen passend gestalten

Die zweite Hälfte der Geschichte

Methoden überschreiben (Zahlungen für einige Mitarbeiter ändern)

Eine Java-Annotation

Methoden aus Klassen und Unterklassen verwenden

9 Neue Objekte entwerfen

Konstruktoren definieren (was es bedeutet, eine Temperatur zu sein)

Was ist eine Temperatur?

Was ist eine Temperaturskala? (Der Java-Typ »enum«)

Also gut, was ist denn nun eine Temperatur?

Was Sie mit einer Temperatur anfangen können

Eine andere Temperatur (32.0) aufrufen – eine Fallstudie

Einige Dinge ändern sich nie

Noch mehr Unterklassen (etwas gegen das Wetter unternehmen)

Für bessere Temperaturen sorgen

Konstruktoren für Unterklassen

Alles anwenden

Der Standardkonstruktor

Ein Konstruktor, der mehr kann

Klassen und Methoden der Java-API

Die Annotation »SuppressWarnings«

Teil IV

Fortgeschrittene Java-Techniken

10 Variablen und Methoden richtig platzieren

Klassen definieren (Was es heißt, ein Baseballspieler zu sein)

Ein anderer Weg, um Zahlen zu formatieren

Die Klasse »Player« verwenden

Neunmal

Eine GUI

Eine Ausnahme von Methode zu Methode weiterreichen

Etwas statisch machen (den Teamdurchschnitt herausfinden)

Warum gibt es da so viel Statisches?

Das statische Initialisierungsprogramm

Den Teamdurchschnitt anzeigen

»static« ist ein alter Hut

Achtung Statisches – mit Vorsicht genießen!

Mit Variablen experimentieren

Eine Variable richtig platzieren

Einer Variablen aufzeigen, wohin sie gehört

Parameter übergeben

Übergabe als Wert

Ein Ergebnis zurückgeben

Übergabe per Referenz

Ein Objekt von einer Methode zurückgeben

Nachtrag

11 Arrays verwenden, um mit Werten zu jonglieren

Bereiten Sie sich gut vor

Ein Array in zwei einfachen Schritten erstellen

Werte speichern

Tabulatoren und anderes

Einen Array-Initialisierer verwenden

Ein Array mit einer erweiterten »for«-Schleife durchlaufen

Suchen

In eine Datei schreiben

Wann eine Datei geschlossen werden muss

Arrays aus Objekten

Die Klasse »Room« verwenden

Und noch ein Weg, um Zahlen schön zu gestalten

Der Bedingungsoperator

Argumente auf der Befehlszeile

Befehlszeilenargumente in einem Java-Programm verwenden

Prüfen, ob die richtige Anzahl an Befehlszeilenparametern
vorhanden ist

12 Sammlungen und Streams verwenden (wenn Arrays nicht
mehr ausreichen)

Hilfe durch Sammelklassen

Eine »ArrayList« verwenden

Generische Typen verwenden

Testen, ob noch mehr Daten vorhanden sind

Einen Iterator verwenden

Die vielen Sammelklassen Javas

Neu in Java 8: funktionale Programmierung

Ein Problem auf die altbewährte Weise lösen

Streams

Lambda-Ausdrücke

Eine Klassifizierung von Lambda-Ausdrücken

Streams und Lambda-Ausdrücke verwenden

Warum sich Sorgen machen?

Methodenreferenzen

13 Auch dann noch gut aussehen, wenn die Dinge eine unerwartete Wendung nehmen

Mit Ausnahmen umgehen

Der Parameter einer »catch«-Klausel

Ausnahmetypen

Wer fängt die Ausnahme ein?

Die Multi-catch-Klausel

Alle Bedenken in den Wind schlagen

Etwas Sinnvolles tun

Unsere Freunde, die guten Ausnahmen

Eine Ausnahme verarbeiten oder den Schwarzen Peter weitergeben

Die Aufgabe mit der Klausel »finally« abschließen«

Ein »try«-Befehl mit Ressourcen

14 Programmteile gemeinsam nutzen lassen

Zugriffsmodifizierer

Klassen, Zugriff und Multipart-Programmierung

Mitglied oder Klassen

Zugriffsmodifizierer für Mitglieder

Eine Zeichnung auf einem Frame platzieren

Die Verzeichnisstruktur

Einen Frame erstellen

Sich heimlich vom ursprünglichen Code absetzen

Standardzugriff

Wieder ins Paket zurückkrabbeln

Geschützter Zugriff

Klassen, die keine Unterklassen sind, in demselben Paket unterbringen

Zugriffsmodifizierer für Java-Klassen

Öffentliche Klassen

Nicht öffentliche Klassen

15 Auf Tastatureingaben und Mausklicks reagieren

Machen Sie weiter ... Klicken Sie auf diese Schaltfläche

Ereignisse und Ereignisbehandlung

Das Java-Interface

Threads

Das Schlüsselwort »this«

Das Innere der Methode »actionPerformed«

Die serialVersionUID

Auf andere Dinge als auf das Anklicken von Schaltflächen reagieren

Innere Klassen erstellen

16 Java-Applets schreiben

Applets-Einmaleins

Auf einen Aufruf warten

Eine öffentliche Klasse

(Noch einmal) die Java-API

Die Dinge in Bewegung versetzen

Die Methoden in einem Applet

Was alles in diese Methoden gehört

Auf Ereignisse in einem Applet reagieren

17 Mit Java Datenbankverbindungen aufbauen und nutzen

JDBC und Java DB

Daten erstellen

SQL-Befehle verwenden

Verbinden und trennen

Daten abfragen

Teil VII

Der Top-Ten-Teil

18 Zehn Wege, um Fehler zu vermeiden

Großbuchstaben dorthin setzen, wo sie hingehören

Aus einer »switch«-Anweisung aussteigen

Werte mit einem doppelten Gleichheitszeichen vergleichen

Einer GUI Komponenten hinzufügen

Listeners für Ereignisbehandlungen hinzufügen

Die benötigten Konstruktoren definieren

Nicht statische Verweise reparieren

Sich in den Grenzen eines Arrays aufhalten

»NullPointerException« vorhersehen

Java bei der Suche nach seinen Dateien helfen

19 Zehn Websites für Java

Die Websites dieses Buches

Aus erster Hand

News, Berichte und Beispielcode

Es gibt nicht nur Englisch

Jedermanns Lieblingssite

Stichwortverzeichnis

Einführung

Java ist klasse. Ich arbeite seit Jahren damit. Ich liebe Java, weil es strukturiert ist. So gut wie alles folgt einfachen Regeln. Diese Regeln können zwar gelegentlich ein wenig bedrohlich wirken, aber eine der Aufgaben dieses Buches ist es, Ihnen dabei zu helfen, diese Regeln zu verstehen. Wenn Sie also Java verwenden und eine Alternative zu den üblichen technikorientierten Taschenbüchern erleben wollen, suchen Sie sich einen gemütlichen Platz, entspannen Sie sich und lesen Sie *Java für Dummies* in der neuesten Auflage.

Wie Sie an dieses Buch herangehen sollten

Ich wünschte, dass ich sagen könnte: »Öffnen Sie das Buch an einer beliebigen Stelle und fangen Sie an, Java-Code zu schreiben. Ergänzen Sie einfach die Lücken im Text und schauen Sie nicht zurück.« In gewisser Weise stimmt das auch, denn Sie können nichts kaputt machen, wenn Sie Java-Code schreiben. Aus diesem Grund sollten Sie sich nicht scheuen, nach Belieben mit Java zu experimentieren.

Aber kommen wir auf den Boden der Tatsachen zurück. Wenn Sie das große Ganze nicht verstehen, ist es schwer, ein Programm zu schreiben. Dies gilt für jede Programmiersprache – also nicht nur für Java. Wenn Sie Code schreiben, ohne genau zu wissen, worum es dabei geht, und wenn dann dieser Code nicht das macht, was er eigentlich sollte, hängen Sie ganz schnell in der Luft.

Ich habe deshalb in diesem Buch die Java-Programmierung in handliche Stücke aufgeteilt. Jedes dieser Stücke entspricht ungefähr einem Kapitel. Sie können an einer beliebigen Stelle einsteigen – [Kapitel 5](#), [Kapitel 10](#) oder anderswo. Sie können sogar in der Mitte eines Kapitels beginnen. Ich habe versucht, die

Beispiele interessant zu gestalten, ohne dass ein Kapitel auf einem anderen aufbaut. Wenn ich einmal eine wichtige Idee eines Kapitels an anderer Stelle wieder aufgreife, binde ich einen Hinweis ein, damit Sie wissen, worum es geht.

Ganz allgemein rate ich Ihnen Folgendes:

- ✓ Wenn Sie etwas schon kennen, vergeuden Sie Ihre Zeit nicht damit, es erneut zu lesen.
- ✓ Wenn Sie etwas besonders interessiert, scheuen Sie sich nicht, an die entsprechenden Stellen zu springen. Sie können bei Bedarf jederzeit wieder einen Blick in ein früheres Kapitel riskieren.

Konventionen, die in diesem Buch verwendet werden

So gut wie jedes technisch orientierte Buch beginnt mit einer Erklärung der verwendeten Schriftarten, und *Java für Dummies* bildet hier keine Ausnahme.

Es folgt nun eine kurze Beschreibung der Schriftarten, die in diesem Buch verwendet werden:

- ✓ Neue Begriffe werden *kursiv* gesetzt.
- ✓ Wenn Sie etwas eingeben müssen, das sich im normalen Text befindet, erscheint es in einer besonderen Schriftart. Zum Beispiel: »Geben Sie in das Textfeld `MeinNeuesProjekt` ein.« Ich verwende diese Schriftart auch für Meldungen, die auf dem Bildschirm erscheinen und für Java-Objekte und -Befehle. Außerdem nutze ich diese Formatierung auch, wenn es um Dateinamen, Bezeichnungen von Verzeichnissen und Adressen von Webseiten (URLs) geht.
- ✓ Java-Code, der als (längere oder kürzere) Listings wiedergegeben wird, wird so gesetzt:

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

```
int weightOfAPerson;
```

- ✓ Wenn Sie Dinge auf Ihrem eigenen Computer eingeben, müssen Sie gegebenenfalls einiges anpassen. So könnte ich Sie etwa auffordern, den folgenden Text zu schreiben:

```
public class EinBeliebigerName
```

Diese kursive Formatierung weist in diesem Fall darauf hin, dass Sie `public class` und einen von Ihnen gewählten, beliebigen Namen eingeben sollen.

Was Sie nicht lesen müssen

Nehmen Sie sich das erste Kapitel oder den ersten Abschnitt vor, in dem etwas steht, was Sie noch nicht kennen, und fangen Sie dort mit dem Lesen an. Wenn Sie es genauso wenig lieben wie ich, Entscheidungen zu fällen, kommen hier ein paar Hinweise, denen Sie folgen können:

- ✓ Wenn Sie bereits wissen, was sich hinter der Spezies Java verbirgt, und wenn Sie wissen, dass Sie Java einsetzen wollen, überspringen Sie [Kapitel 1](#) und gehen Sie geradewegs zu [Kapitel 2](#). Sie können mir glauben, dass mir das nichts ausmacht.
- ✓ Wenn Sie bereits wissen, wie Sie ein Java-Programm ans Laufen bekommen, und wenn Sie an dem, was dabei im Hintergrund passiert, kein Interesse haben, überspringen Sie [Kapitel 2](#) und beginnen Sie gleich mit [Kapitel 3](#).
- ✓ Wenn Sie beruflich Programme schreiben, dabei aber eine andere Sprache als C oder C++ verwenden, beginnen Sie mit [Kapitel 2](#) oder 3. Wenn Sie [Kapitel 5](#) und 6 lesen, könnten Sie der Meinung sein, dass sie zu leicht für Sie sind, aber spätestens in [Kapitel 7](#) tauchen auch Sie tief in die Materie ein.
- ✓ Wenn Sie zum Lebensunterhalt Programme in C (nicht C++) schreiben, beginnen Sie mit den [Kapiteln 2](#), 3 und 4 und überspringen Sie nur die [Kapitel 5](#) und 6.

- ✓ Wenn Sie zum Lebensunterhalt Programme in C++ schreiben, werfen Sie einen Blick auf die [Kapitel 2](#) und 3, überspringen Sie die [Kapitel 4](#) bis 6 und beginnen Sie ein ernsthaftes Lesen mit [Kapitel 7](#). (Java unterscheidet sich beim Umgang mit Klassen und Objekten ein wenig von C++.)
- ✓ Wenn Sie zum Lebensunterhalt Programme in Java schreiben, besuchen Sie mich zu Hause und helfen Sie mir dabei, die nächste Auflage von *Java für Dummies* zu schreiben.

Wenn Sie Randbemerkungen und die technischen Informationen überspringen wollen, hindere ich Sie nicht daran, und wenn Sie sogar auf alle Informationen verzichten möchten, haben Sie die Freiheit, dies zu tun.

Ein paar einfache Annahmen

Ich setze in diesem Buch bei Ihnen, dem Leser, einiges voraus. Wenn eine dieser Annahmen nicht richtig ist, geht das noch in Ordnung. Wenn aber alle Annahmen falsch sind ... nun, dann sollten Sie das Buch trotzdem kaufen.

- ✓ **Ich gehe davon aus, dass Sie Zugriff auf einen Computer haben.** Die gute Nachricht ist: Sie können fast den gesamten Code, den dieses Buch enthält, auf so gut wie jedem Computer ablaufen lassen. Die einzigen Computer, die Sie nicht verwenden können, sind altherwürdige Geräte, die vor zehn Jahren (plus/minus ein paar Jahre) modern gewesen sind.
- ✓ **Ich gehe davon aus, dass Sie sich mit den normalen Menüstrukturen und Dialogfeldern Ihres Computers auskennen.** Sie müssen keine besonderen Erfahrungen in Windows, Linux oder Macintosh haben, aber Sie sollten in der Lage sein, ein Programm zu starten, eine Datei zu finden, eine Datei in einem bestimmten Verzeichnis abzulegen ... also diese Art von Dingen zu erledigen. Wenn Sie sich mit dem Stoff dieses Buches beschäftigen, benutzen Sie die meiste Zeit Ihre Tastatur, statt mit der Maus zu zeigen und zu klicken.

In den seltenen Fällen, bei denen Sie etwas ziehen und fallen lassen, ausschneiden und einfügen oder anschließen und verwenden müssen, leite ich Sie umsichtig durch die einzelnen Schritte. Es kann aber vorkommen, dass Ihr Computer auf eine von Milliarden möglichen Arten eingerichtet worden ist, die dafür sorgt, dass meine Anleitung nicht ganz auf Ihre spezielle Situation zugeschnitten ist. Auch in solch einem Fall sollten Sie versuchen, den Schritten in diesem Buch zu folgen. Wenn die Schritte nicht passen, sollten Sie ein Buch zurate ziehen, das für Ihr System maßgeschneidert ist.

✓ **Ich gehe davon aus, dass Sie logisch denken können.**

Mehr gehört zum Programmieren in Java nicht dazu – logisches Denken. Wenn Sie in der Lage sind, logisch zu denken, haben Sie es geschafft. Wenn Sie daran zweifeln, dass Sie logisch denken können, lesen Sie weiter. Sie werden höchstwahrscheinlich angenehm überrascht sein.

✓ **Ich setze bei Ihnen nur kaum (oder überhaupt keine)**

Programmierkenntnisse voraus. Beim Schreiben dieses Buches habe ich das Unmögliche versucht. Ich wollte dieses Buch für erfahrene Programmierer interessant machen, ohne dabei diejenigen aus den Augen zu verlieren, die nur wenige oder keine Programmierkenntnisse besitzen. Dies bedeutet, dass ich bei Ihnen keinen irgendwie gearteten programmiertechnischen Hintergrund voraussetze. Wenn Sie noch nie im Leben eine Schleife angelegt oder ein Array indexiert haben, ist das mehr als in Ordnung.

Wenn Sie sich aber mit diesen Dingen (zum Beispiel in Visual Basic, COBOL oder C++) beschäftigt haben, werden Sie in Java einige interessante Wendungen beim Handlungsablauf entdecken. Die Entwickler von Java haben die besten Ideen der objektorientierten Programmierung genommen, sie optimiert und das Herangehen an Probleme geradlinig gestaltet. Sie werden in Java auf viele neue, zum Nachdenken anregende Funktionen stoßen. Und Sie werden feststellen, dass Ihnen viele davon als ganz selbstverständlich

vorkommen werden. Auf jeden Fall wird es Ihnen Spaß machen, Java einzusetzen.

Wie dieses Buch aufgebaut ist

Dieses Buch besteht aus Unterabschnitten, die zu Abschnitten gruppiert worden sind, die wiederum Kapitel bilden, die letztendlich in fünf Teilen zusammengefasst wurden. (Wenn Sie ein Buch schreiben, sollten Sie den Aufbau des Buches ziemlich gut kennen. Nach Monaten des Schreibens werden Sie sich selbst dabei ertappen, dass Sie nachts in Abschnitten und Kapiteln träumen.) Das Buch besteht aus diesen Teilen:

Teil I: Los geht's

Dieser Teil enthält eine vollständige Einführung in Java. Hier gibt es Dinge wie »Was ist Java?« und ein Kapitel für einen Schnellstart – [Kapitel 3](#). Sie untersuchen in [Kapitel 3](#) die zentralen technischen Gedanken und analysieren ein einfaches Programm.

Teil II: Eigene Java-Programme schreiben

Die [Kapitel 4](#) bis 6 behandeln Grundsätzliches. Diese Kapitel beschreiben die Dinge, die Sie wissen müssen, damit Ihr Computer bei Laune bleibt.

Wenn Sie bereits in Visual Basic, C++ oder einer anderen Sprache programmiert haben, kommt Ihnen vielleicht einiges aus Teil II bekannt vor. In diesem Fall können Sie einige Abschnitte überspringen oder den Stoff einfach nur querlesen. Seien Sie dabei aber nicht zu schnell. Java unterscheidet sich in einigen Dingen von anderen Programmiersprachen, wobei es hier ganz besonders um das geht, was in [Kapitel 4](#) behandelt wird.

Teil III: Das große Ganze im Auge behalten: die objektorientierte

Programmierung

Teil III enthält einige meiner Lieblingskapitel. Dieser Teil behandelt das überaus wichtige Thema der objektorientierten Programmierung. Sie finden in diesen Kapiteln heraus, wie Sie zu großen Problemen Lösungen finden. (Klar, die Beispiele in diesen Kapiteln sind nichts Besonderes, aber sie enthalten besondere Ideen.) Sie entdecken Schritt für Schritt, wie Klassen entworfen, vorhandene Klassen erneut verwendet und wie Objekte konstruiert werden.

Haben Sie schon einmal eines dieser Bücher gelesen, in denen die objektorientierte Programmierung mit ungenauen, allgemeinen Begriffen erklärt wird? Ich bin stolz darauf, behaupten zu können, dass *Java für Dummies* anders ist. Ich erkläre in diesem Buch jedes Konzept mithilfe eines einfachen, aber passenden Programmbeispiels.

Teil IV: Fortgeschrittene Java-Techniken

Wenn Sie Java ausprobiert und Lust auf mehr bekommen haben, finden Sie in diesem Teil des Buches das, was Sie dazu brauchen. Die Kapitel dieses Teils beschäftigen sich mit Details – den Dingen, die Sie leicht übersehen, wenn Sie sich zum ersten Mal mit Java beschäftigen. Wenn Sie die früheren Teile des Buches gelesen und ein paar eigene Programme geschrieben haben, können Sie ein wenig tiefer in die Materie eintauchen, indem Sie Teil IV lesen.

Teil V: Der Top-Ten-Teil

Der Top-Ten-Teil ist ein kleiner Laden für Java-Schmankerl. Hier finden Sie Listen – Listen mit Tipps, um Fehler zu vermeiden, Listen, um Hilfe zu finden, und Listen mit anderen interessanten Dingen.

Symbole, die in diesem Buch verwendet werden

Wenn Sie mich beim Schreiben dieses Buches hätten beobachten können, hätten Sie gesehen, wie ich an meinem Computer saß und mit mir selbst geredet habe. Ich habe mir in meinem Kopf jeden Satz vorgelesen. Die meisten Sätze habe ich ein paar Mal gemurmelt. Wenn dann ein zusätzlicher Gedanke auftauchte, ein ergänzender Kommentar oder etwas, das nicht in den normalen Ablauf des Textes passte, habe ich gedankenverloren mit dem Kopf gewackelt.

Natürlich können Sie dieses Kopfschütteln im gedruckten Buch nicht sehen. Ich muss also auf etwas anderes zurückgreifen, um diese Gedanken irgendwie darzustellen. Ich mache dies mit Symbolen. Wenn Sie ein Tipp- oder ein Erinnerung-Symbol sehen, wissen Sie, dass ich einen kleinen Abstecher mache.

Hier eine Liste mit den Symbolen, die ich in diesem Buch verwende:



Ein Tipp ist eine kleine, aber feine zusätzliche Information – etwas Nützliches, das Sie in anderen Büchern vielleicht nicht finden.



Jeder Mensch macht Fehler. Auch mir ist das im Laufe der Zeit immer wieder einmal passiert. Wenn ich glaube, dass eine Situation ganz besonders dazu verleitet, Fehler zu machen, weise ich mit dem Symbol »Warnung« darauf hin.



Frage: Was ist stärker als ein Tipp, aber nicht so stark wie eine Warnung?

Antwort: Ein Symbol »Erinnerung«.



Gelegentlich begegne ich technischen Leckerbissen. Ein solcher Leckerbissen kann Ihnen dabei helfen zu verstehen, was die Menschen im Hintergrund (diejenigen, die Java entwickelt haben) gedacht haben. Sie müssen Informationen dieser Art nicht lesen, aber vielleicht sind sie nützlich. Und vielleicht helfen Ihnen diese zusätzlichen Informationen, wenn Sie andere (spezialisiertere) Bücher über Java lesen.



Dieses Symbol soll Ihre Aufmerksamkeit auf nützliches Material lenken, das Sie im Web finden können. (Sie müssen nicht lange auf dieses Symbol warten. Ich verwende eines am Ende dieser Einführung!)

Anmerkung der Übersetzerin

Die Entwicklungsumgebung, die in diesem Buch vorgestellt wird, ist – genau wie die Programmiersprache Java – englischsprachig. Aus diesem Grund werden Sie beim Lesen auch immer wieder auf englische Begriffe stoßen, wobei wir uns bemühen, diese zumindest bei ihrer ersten Verwendung auch auf Deutsch vorzustellen.

Zu diesem Buch gehören Beispieldateien, die kapitelweise den Quellcode der vorgestellten Programme beziehungsweise Programmstückchen enthalten. Um möglichst nah an diesen vom Autor vorgegebenen Beispieldateien zu bleiben, wurden auch in der Übersetzung Namen und Bezeichnungen aus dem Original übernommen und nicht generell eingedeutscht.

Wie es weitergeht

Wenn Sie bis hierher gekommen sind, sind Sie so weit, Näheres über das Entwickeln von Anwendungen mit Java zu erfahren. Betrachten Sie mich (den Autor) als Ihren Führer, Ihren Gastgeber, Ihren persönlichen Assistenten. Ich mache alles, was

in meiner Macht steht, um die Dinge interessant zu halten und – wichtiger noch – Ihnen dabei zu helfen, sie zu verstehen.



Wenn Ihnen das, was Sie lesen, gefällt, oder wenn Sie Hilfe benötigen, senden Sie mir eine E-Mail an die Adresse, die ich nur für dieses Buch eingerichtet habe:

`JavaForDummies@allmycode.com`. Wenn E-Mail oder Chat nicht zu Ihren Favoriten zählt, können Sie mich auch über Twitter (@allmycode) und Facebook (/allmycode) erreichen. Denken Sie aber daran, dass Kontakte unbedingt in englischer Sprache erfolgen müssen. Und denken Sie an Aktualisierungen. Die neuesten befinden sich auf der von mir angelegten Website zu diesem Buch

(www.allmycode.com/javafordummies), während Sie die Demodateien zur Übersetzung unter www.wiley-vch.de/publish/dt/books/ISBN3-527-71070-1 herunterladen können.

Über den Autor

Barry Burd beendete sein Studium der Computerwissenschaften an der Rutgers University mit einem Master of Science und erlangte an der University of Illinois den PhD (Doktorgrad) in Mathematik. Während seiner Lehrtätigkeit als Assistent in Champaign-Urbana wurde er von den Studenten fünf Mal in die universitätsweite Bestenliste aller Lehrkräfte gewählt.

Dr. Burd arbeitet seit 1980 als Professor am Department of Mathematics and Computer Science der Drew University in Madison, New Jersey. Wenn er nicht gerade lehrt, leitet Dr. Burd Kurse für Programmierer aus Handel, Beratung und Industrie. Außerdem hat er in den USA, in Europa, Australien und Asien auf Konferenzen Vorträge gehalten. Er hat diverse Artikel und Bücher geschrieben, zu denen die bei Wiley erschienenen Titel *Java für Dummies*, *Mit Java programmieren lernen für Dummies* und *Android Application Development All-in-One For Dummies* gehören.

Dr. Burd lebt mit seiner Frau und zwei Kindern (die beide in den 20ern und sehr selbstständig sind) in Madison, New Jersey. In seiner Freizeit liebt er es, ein »Workaholic« zu sein.

Teil I

Los geht's



In diesem Teil . . .

- ✓ Sie finden heraus, welche Werkzeuge Sie für das Entwickeln von Java-Programmen benötigen.
- ✓ Sie finden heraus, wie Java in die heutige technologische Landschaft passt.
- ✓ Sie sehen Ihr erstes vollständiges Java-Programm.

1

Alles über Java

In diesem Kapitel

- ▶ Was Java ist
 - ▶ Woher Java stammt
 - ▶ Warum Java so cool ist
 - ▶ Wie Sie sich in der objektorientierten Programmierung zurechtfinden
-

Sie können von Computern halten, was Sie wollen, soweit es mich betrifft, sprechen zwei gute Gründe für Computer:

✓ **Wenn Computer ihre Arbeit erledigen, gibt es keinen Widerspruch, sie kennen keinen Stress, empfinden keine Langeweile und ermüden nie.** Computer sind unsere elektronischen Sklaven. Mein Computer hat schon sieben Tage die Woche 24 Stunden am Tag gearbeitet und Berechnungen für Cosmology durchgeführt. Dabei handelt es sich um ein verteiltes Berechnungsprojekt, um Modelle herauszufinden, die das Universum beschreiben. Hat mir mein Computer leidgetan, weil er so hart arbeiten musste? Hat sich der Computer beklagt? Meldet mich der Computer bei der Gewerkschaft? Nein.

Ich kann Forderungen stellen, dem Computer Anweisungen erteilen und mit der Peitsche knallen. Sollte ich mich deshalb schuldig fühlen? Nein, natürlich nicht.

✓ **Computer bewegen Ideen, kein Papier.** Vor noch nicht allzu langer Zeit mussten Sie einen Boten anheuern, wenn Sie jemandem eine Nachricht zukommen lassen wollten. Dieser Bote stieg auf sein Pferd und lieferte Ihre Nachricht persönlich

ab. Die Nachricht stand auf einem Blatt Papier, einem Pergament, einer Tontafel oder was gerade verfügbar war.

Aus heutiger Sicht scheint dieser Vorgang unwirklich zu sein, aber das kommt nur daher, dass Sie und ich im elektronischen Zeitalter bequem vor einem Computer sitzen. Nachrichten sind Ideen und physische Dinge wie Tinte, Papier und Pferde haben nichts oder nur wenig mit echten Ideen zu tun; sie sind nur die Überbringer der Ideen (selbst wenn die Menschen sie jahrhundertlang als Träger von Ideen benutzt haben). Auf jeden Fall gilt, dass Ideen papierlos sind und ohne Pferde und Boten auskommen.

Das Schöne an Computern ist, dass sie Ideen effizient übertragen. Sie befördern nichts als die Ideen, ein paar Photonen und etwas Strom. Und sie erledigen dies ordentlich, ohne zu jammern und ohne zusätzliches Gepäck.

Wenn Sie anfangen, sich ernsthaft mit Ideen zu beschäftigen, geschieht etwas wirklich Angenehmes. Plötzlich verschwindet ein Großteil des Mehraufwands. Statt Papier und Bäume stoßen Sie Zahlen und Konzepte an. Sie können Dinge ohne den Mehraufwand viel schneller erledigen und sind in der Lage, sich mit viel komplexeren Sachen zu beschäftigen als je zuvor.

Was Sie mit Java machen können

Es wäre nun richtig klasse, wenn diese Vielfalt einfach so vorhanden wäre, aber unglücklicherweise ist das nicht der Fall. Jemand muss schwer nachdenken und entscheiden, was der Computer genau machen soll. Nach dem Nachdenken muss sich jemand hinsetzen und Anweisungen für den Computer schreiben, denen dieser zu folgen hat.

Der heutige Stand der Dinge verbietet es leider, diese Anweisungen in Deutsch oder einer anderen gesprochenen Sprache zu schreiben. Science-Fiction lebt auch von

Geschichten, in denen Menschen Robotern einfache Dinge sagen, die zu desaströsen, unerwarteten Ergebnissen führen. Deutsch und andere Sprachen eignen sich aus mehreren Gründen nicht für eine Kommunikation mit Computern:

- ✓ **Deutsche Sätze können mehrdeutig sein.** »Engländer tragen zu feierlichen Anlässen gerne eine Melone.«
- ✓ **Es ist schwierig, im Deutschen einen komplizierten Befehl in seine logischen Einzelteile zu zerlegen.** »Verbinden Sie Flansch A mit Protuberanz B und sorgen Sie dafür, dass nur der äußerste Ring von Flansch A mit dem größeren Ende der Protuberanz B verbunden wird, während der mittlere und der innere Ring des Flansches A an den Dichtungsring C gehören.«
- ✓ **Ein deutscher Satz schleppt viel Ballast mit sich herum.** »Der Satz enthält überflüssige Wörter.«
- ✓ **Es ist manchmal schwierig, Deutsch zu interpretieren.** »Als Teil des Verwertungsvertrages zwischen der Firma Rubbeldiwubbel GmbH & Co. KG (›Rubbek‹) und dem Autor (›Barry Burd‹) zahlt Rubbel den Betrag von eintausendzweihundertfünfsiebzug Euro und siebzehn Cent (1.275,17 EUR) an Barry Burd als erste Abschlagszahlung für den Auftrag 631-A19-8612.«

Um einem Computer mitzuteilen, was er machen soll, müssen Sie eine besondere Sprache sprechen und in dieser Sprache knappe, unmissverständliche Anweisungen (auch *Befehle* genannt) schreiben. Eine Sprache dieser Art wird *Programmiersprache* genannt. Eine Folge von Anweisungen, die in solch einer Sprache geschrieben worden ist, wird *Programm* genannt. Eine andere Bezeichnung für ein Programm ist *Software* oder *Code*. Und so sieht Code aus, wenn er in Java geschrieben worden ist:

```
public class PayBarry {  
    public static void main(String args[]) {  
        double checkAmount = 1257.63;  
        System.out.print("Zahlen Sie an ");  
        System.out.print("Dr. Barry Burd ");  
        System.out.print("€");  
    }  
}
```

```
        System.out.println(checkAmount);  
    }  
}
```

Warum Sie Java verwenden sollten

Es gibt einen Grund zum Feiern. Sie haben sich gerade ein Exemplar von *Java für Dummies* besorgt und Sie lesen [Kapitel 1](#). Bei diesem Tempo werden Sie in Nullkommanichts zu einem Experten in der Java-Programmierung. Genießen Sie deshalb Ihren voraussichtlichen Erfolg, indem Sie eine große Party veranstalten.

Als Vorbereitung für diese Party will ich einen Kuchen backen. Da ich bequem bin, verwende ich eine fertige Backmischung. Mal sehen ... fügen Sie zur Mischung Wasser hinzu, dann Butter und Eier ... Halt, einen Moment! Ich habe mir gerade die Liste mit den Zutaten angesehen. Was ist MSG? Und was hat es mit Propylenglykol auf sich? Das wird doch auch als Frostschutzmittel verwendet, oder?

Ich muss meine Pläne ändern und den Kuchen selbst machen. Das ist zwar ein wenig umständlicher, aber so bekomme ich genau das, was ich haben will.

Computerprogramme arbeiten auf die gleiche Weise. Sie können ein Programm verwenden, das von jemandem stammt, oder Sie schreiben ein eigenes Programm. Wenn Sie sich für die erste Variante entscheiden, müssen Sie das nutzen, was Sie bekommen. Wenn Sie Ihr eigenes Programm schreiben, können Sie es auf Ihre Bedürfnisse hin maßschneidern.

Das Schreiben von Computerprogrammen ist zu einer großen, weltweiten Industrie geworden. Firmen machen es, freiberufliche Profis machen es, und es ist zum Hobby vieler geworden; alle möglichen Menschen machen es. Ganz normale große Unternehmen haben Teams, Abteilungen und Bereiche, die

Programme für das Unternehmen schreiben. Und auch Sie können Programme schreiben – für sich selbst, zum Lebensunterhalt oder nur, weil es Spaß macht. Neueste Schätzungen besagen, dass die Zahl der Codezeilen, die allein in den Vereinigten Staaten tagtäglich von Entwicklern geschrieben werden, die Zahl der Methan-Moleküle auf dem Planeten Jupiter überschreitet (wobei die entsprechende Schätzung von mir selbst stammt). Stellen Sie sich vor, was Sie alles mit einem Computer anstellen können. Und mit dem entsprechenden Zeitaufwand sollten Sie in der Lage sein, dafür Ihr eigenes Programm zu schreiben. (Wobei der »entsprechende Zeitaufwand« ziemlich groß sein kann, worum es hier aber nicht geht. Viele interessante Programme können in wenigen Stunden oder sogar Minuten geschrieben werden.)

Einen Überblick erhalten: Wie sich Java einordnen lässt

Hier ein kurzer Abriss der modernen Programmierung auf Computern:

✓ **1954 – 1957: FORTRAN wird entwickelt.**

FORTRAN war die erste moderne Programmiersprache für Computer. Es entpuppte sich für die wissenschaftliche Programmierung als echter Renner. Jahrelang war FORTRAN weltweit die führende Programmiersprache.

✓ **1959: Grace Hopper von Remington Rand entwickelt die Programmiersprache COBOL.**

Der Buchstabe *B* steht in COBOL für *Business*, und bei COBOL geht es fast ausschließlich um Geschäftliches. Die primäre Aufgabe dieser Sprache ist es, einen Datensatz nach dem anderen, einen Kunden nach dem anderen oder einen Mitarbeiter nach dem anderen zu verarbeiten.

COBOL wurde innerhalb weniger Jahre die am häufigsten eingesetzte Sprache für die Verarbeitung geschäftlicher Daten.

COBOL stellt selbst heute noch einen großen Teil der programmierenden Industrie dar.

✓ **1972: Dennis Ritchie von AT&T Bell Labs entwickelt die Programmiersprache C.**

Das Aussehen der Beispielprogramme in diesem Buch hat ihren Ursprung in der Programmiersprache C. Code, der in C geschrieben wird, verwendet geschweifte Klammern, `if`- und `for`-Anweisungen und so weiter.

Wenn es um die Leistungsfähigkeit geht, können Sie C verwenden, um dieselben Probleme zu lösen, die sich mit FORTRAN, Java oder einer anderen modernen Programmiersprache beseitigen lassen. (Sie können auch in COBOL einen wissenschaftlichen Taschenrechner schreiben, aber das wäre ein Akt, den Sie so schnell nicht vergessen werden.) Der Unterschied zwischen den einzelnen Programmiersprachen ist nicht die Leistungsfähigkeit. Der Unterschied liegt darin, wie einfach und bedienerfreundlich eine Sprache ist.

✓ **1986: Bjarne Stroustrup (wieder von den AT&T Bell Labs) entwickelt C++.**

Die Sprache C++ unterstützt, anders als ihr Vorgänger C, objektorientierte Programmierung. Dies stellte einen riesigen Schritt vorwärts dar. (Siehe auch den nächsten Abschnitt in diesem Kapitel.)

✓ **23. Mai 1995: Sun Microsystems veröffentlicht die erste offizielle Version der Programmiersprache Java.**

Java verbessert die Konzepte von C++. Javas Philosophie – »Einmal schreiben, überall ablaufen lassen« – machte die Sprache zum idealen Werkzeug, um Code über das Internet zu verteilen.

Außerdem ist Java eine Programmiersprache, die sich für so gut wie alle Zwecke einsetzen lässt. Sie können mit Java Anwendungen mit Fenstern schreiben, Datenbanken erstellen und auswerten, Handheld-Geräte kontrollieren und vieles

mehr machen. Die Programmiersprache Java erreichte innerhalb von nur fünf kurzen Jahren 2,5 Millionen Entwickler weltweit. (Ich weiß das, denn ich besitze ein entsprechendes T-Shirt, das das beweist.)

- ✓ **November 2000: Das College Board gibt bekannt, dass ab 2003 alle Computer-Science-Advanced-Placement-Examen auf Java basieren.**

Sie wollen wissen, was die kleinen Rotznasen in der Highschool nur noch lernen? Sie haben es erraten – Java.

- ✓ **2002: Microsoft stellt eine neue Sprache vor, die den Namen C# trägt.**

Viele der Funktionen von C# stammen direkt von Java-Funktionen ab.

- ✓ **Juni 2004: Sys-Con Media (java.sys-con.com/node/48507) berichtet, dass die Nachfrage nach Java-Programmierern um 50 Prozent größer ist als die nach C++-Programmierern.**

Aber das ist noch nicht alles. Die Nachfrage nach Java-Programmierern schlägt die Nachfrage nach C++- und C#-Programmierern zusammen um acht Prozent. Java-Programmierer haben eine um 180 Prozent größere Chance auf einen Job als Programmierer, die mit VB (Visual Basic) arbeiten.

- ✓ **2007: Google führt Java als primäre Sprache für das Erstellen von Apps auf mobilen Android-Geräten ein.**

- ✓ **Januar 2010: Die Oracle Corporation kauft Sun Microsystems auf, was dazu führt, dass Java in die Produkte der Oracle-Familie einzieht.**

- ✓ **Juni 2010: eWeek führt Java zum ersten Mal in seiner Liste der zehn wichtigsten Programme auf, die Sie Ihren Job behalten lassen**

(www.eweek.com/c/a/Application-Development/Top-10Programming-Languages-to-Keep-You-Employed-719257).

✓ **August 2013: Java wird auf mehr als 1,2 Milliarden Desktopcomputern eingesetzt** (java.com/en/about).
Android-Java läuft auf mehr als 250 Millionen mobilen Telefonen (www.mobilededicemanager.com/mobile-device-statistics/250-million-android-devices-in-use). Java sorgt bei allen Blu-ray-Playern dafür, dass Interaktionen mit den Geräten möglich sind. Java ist laut dem Index der TIOBE Programming Community die beliebteste Programmiersprache (www.tiobe.com/index.php/content/paperinfo/tpci).
i).

Objektorientierte Programmierung (OOP)

Es ist drei Uhr morgens. Ich träume von der Geschichtsstunde, die ich in der Highschool verpasst habe. Der Lehrer schreit mich an: »Du hast zwei Tage, um dich auf die Abschlussprüfung vorzubereiten, aber daran wirst du dich nicht erinnern. Du wirst es vergessen und dich schuldig fühlen.«

Plötzlich klingelt das Telefon. Ich werde abrupt aus dem tiefsten Schlaf gerissen. (Klar, ich liebe es nicht unbedingt, vom Geschichtsunterricht zu träumen, aber noch weniger liebe ich es, aus dem Schlaf geholt zu werden.) Als Erstes ließ ich das Telefon auf den Boden fallen. Nachdem ich eine Weile herumgefummelt hatte, um es wiederzufinden, ließ ich ein mürrisches »Hallo, wer ist da?« hören. Eine Stimme antwortete: »Ich bin Reporter bei der *New York Times*. Ich schreibe einen Artikel über Java und ich muss alles über diese Programmiersprache wissen – in einem kurzen Satz.«

Ich bin noch zu benebelt. Ich kann nicht denken. Ich sage also das, was mir gerade einfällt, und schlafe dann weiter.

Am nächsten Morgen kann ich mich kaum an die Unterhaltung erinnern. Wenn ich ehrlich bin, erinnere ich mich überhaupt nicht

mehr daran, wie ich dessen Fragen beantwortet habe. Habe ich dem Reporter gesagt, was er mit seinem Artikel über Java machen soll?

Ich ziehe mich an und gehe zur Zufahrt meines Hauses. Während ich die Morgenzeitung aufhebe, fällt mein Blick auf die riesengroße Schlagzeile:

Burd nennt Java »Eine großartige objektorientierte Sprache«

Objektorientierte Sprachen

Java ist objektorientiert. Was heißt das? Anders als Sprachen wie FORTRAN, das sich darauf konzentriert, dem Computer befehlende »Mache dies/Mache das«-Anweisungen zu geben, richten objektorientierte Sprachen ihr Hauptaugenmerk auf Daten. Natürlich sagen auch objektorientierte Sprachen einem Computer, was er zu tun hat. Sie beginnen aber mit dem Strukturieren der Daten, und die Befehle kommen dann später.

Objektorientierte Sprachen sind besser als »Mache dies/Mache das«-Sprachen, weil sie Daten auf eine Weise strukturieren, die es dann zulässt, viele Dinge mit diesen Daten anzustellen. Um die Daten zu modifizieren, können Sie auf das aufbauen, was Sie bereits haben, anstatt erst einmal alles zu verschrotten, was Sie bisher entwickelt haben, und jedes Mal, wenn Sie etwas Neues benötigen, wieder ganz von vorn anzufangen. Obwohl Programmierer in der Regel clevere Menschen sind, dauerte es doch eine Weile, bis sie das herausbekommen haben. Wenn Sie die ganze Geschichte erfahren wollen, schauen Sie sich den Kasten *Der kurvenreiche Weg von FORTRAN zu Java* an.

Der kurvenreiche Weg von FORTRAN zu Java

Mitte der 1950er entwickelte eine Gruppe von Leuten eine Programmiersprache mit dem Namen FORTRAN. Es war eine gute Sprache, aber sie basierte auf der Idee, dass dem Computer direkte, befehlende Anweisungen übermittelt werden. »Computer, mache dies. Und dann,

Computer, mache das.« (Natürlich waren die Anweisungen in einem echten FORTRAN-Programm viel genauer als »Mache dies« oder »Mache das«.)

In den folgenden Jahren entwickelten Teams viele neue Computersprachen, von denen viele das »Mache dies/Mache das«-Modell von FORTRAN kopiert haben. Eine der bekannteren »Mache dies/Mache das«-Sprachen war die Ein-Buchstabe-Sprache C. Natürlich gab es im »Mache dies/Mache das«-Lager auch einige Abtrünnige. Programmierer haben in Sprachen wie SIMULA und Smalltalk die befehlenden »Mache dies/Mache das«-Anweisungen in den Hintergrund verlagert und sich auf Beschreibungen der Daten konzentriert. Bei diesen Sprachen stehen Sie nicht auf und sagen: »Drucke eine Liste aller zweifelhaften Konten.« Stattdessen beginnen Sie, indem Sie sagen: »Dies hier ist das, was Konto bedeutet. Ein Konto hat einen Namen und einen Kontostand.« Und dann sagen Sie: »Und so wird ein Konto gefragt, ob es ein zweifelhaftes Konto ist.« Plötzlich werden die Daten zum König. Ein Konto war ein Ding, das einen Namen, einen Kontostand und einen Weg besaß, über den es Ihnen mitteilen konnte, ob es ein zweifelhaftes Konto war.

Sprachen, deren Hauptaugenmerk auf den Daten liegt, werden objektorientierte Programmiersprachen genannt. Diese objektorientierten Sprachen erzeugen ausgezeichnete Programmierwerkzeuge.

Der Grund dafür:

- ✓ Wenn Sie zuerst an die Daten denken, werden Sie zu einem guten Programmierer.
- ✓ Sie können die Beschreibung der Daten erweitern und immer wieder erneut verwenden. Wenn Sie versuchen, alten FORTRAN-Programmen neue Tricks beizubringen, zeigen diese alten Programme, wie empfindlich sie sein können, denn sie gehen in die Brüche.

In den 1970ern wurden objektorientierte Sprachen wie SIMULA und Smalltalk in den Computerzeitschriften, die von Hobbyprogrammierern gelesen wurden, beerdigt. In der Zwischenzeit vermehrten sich Sprachen, die auf dem alten FORTRAN-Modell basierten, wie die Kaninchen.

1986 entwickelte ein Typ mit dem Namen Bjarne Stroustrup eine Sprache, die er C++ nannte. Diese Sprache wurde sehr beliebt, weil sie die Terminologie der alten Sprache C mit der besseren objektorientierten Struktur mischte. Viele Firmen wendeten dem alten Stil der FORTRAN/C-Programmierung den Rücken zu und machten C++ zu ihrem Standard.

Aber C++ hatte eine Schwachstelle. Wenn Sie C++ verwendeten, waren Sie in der Lage, alle objektorientierten Funktionen zu umgehen und ein Programm zu schreiben, in dem Sie auf den alten FORTRAN/C-Programmierstil

zurückgriffen. Wenn Sie anfangen, ein Buchhaltungsprogramm zu schreiben, konnten Sie sich für einen von zwei Wegen entscheiden:

- ✓ Sie konnten damit anfangen, dass Sie dem Computer direkt »Mache dies/Mache das«-Anweisungen erteilten, indem Sie ihm die mathematische Entsprechung von »Drucke eine Liste aller zweifelhaften Konten und das schnell!« vorlegten.
- ✓ Sie konnten objektorientiert vorgehen und anfangen, indem Sie beschrieben, was es heißt, ein Konto zu sein.

Einige waren der Meinung, dass C++ das Beste aus zwei Welten anbietet, während andere argumentierten, dass die erste Welt (FORTRAN und C) nichts in der modernen Programmierung zu suchen hätte. Wenn Sie einem Programmierer die Möglichkeit geben, Code auf eine von zwei Weisen zu schreiben, würde sich der Programmierer zu oft für den falschen Weg entscheiden.

1995 entwickelte dann James Gosling von Sun Microsystems eine Sprache mit dem Namen *Java*. Als er sich mit Java beschäftigte, griff Gosling auf das Erscheinungsbild von C++ zurück. Aber Gosling nahm die meisten »Mache dies/Mache das«-Funktionen von C++ und warf sie auf den Müll. Dann fügte er Funktionen hinzu, die die Entwicklung von Objekten geschmeidiger machten und erleichterten. Gosling entwickelte alles in allem eine Sprache, deren objektorientierte Philosophie unverfälscht und sauber ist. Wenn Sie in Java programmieren, haben Sie keine andere Wahl, als mit Objekten zu arbeiten. Und so sollte es auch sein.

Objekte und ihre Klassen

In einer objektorientierten Sprache verwenden Sie Objekte *und* Klassen, um Ihre Daten zu ordnen.

Stellen Sie sich vor, dass Sie ein Computerprogramm schreiben, dass die (noch im Bau befindlichen) Häuser einer neuen Wohnanlage im Auge behält. Die Häuser unterscheiden sich nur leicht voneinander. Jedes Haus hat einen unverwechselbaren Außenanstrich, einen Innenanstrich, einen Küchenschrank in einem besonderen Stil und so weiter. In Ihrem objektorientierten Programm bildet jedes Haus ein Objekt.

Aber Objekte sind noch nicht alles. Auch wenn sich die Häuser leicht voneinander unterscheiden, so haben sie doch eine Liste

mit Merkmalen gemeinsam. So besitzt zum Beispiel jedes Haus ein Merkmal, das *Außenanstrich* genannt wird. Jedes Haus hat ein Merkmal, das *Stil des Küchenschrank*s heißt. Sie benötigen in Ihrem objektorientierten Programm ein Stammbblatt, das alle Merkmale enthält, die ein Haus besitzen kann. Diese zentrale Liste aller Merkmale wird *Klasse* genannt.

So, nun wissen Sie es. Objektorientierte Programmierung trägt einen falschen Namen. Sie müsste eigentlich *Programmierung mit Klassen und Objekten* heißen.

Beachten Sie, dass ich das Wort *Klassen* an die erste Stelle gesetzt habe. Wie komme ich dazu, mich so etwas zu trauen? Nun, vielleicht bin ich doch nicht verrückt. Erinnern Sie sich noch einmal an die Häuser, die sich noch im Bau befinden. Irgendwo gibt es auf dem Bauplatz in einem Container ein Stammbblatt der Merkmale, das auch *Bauplan* genannt wird. Der Bauplan eines Architekten kann mit der Klasse eines objektorientierten Programmierers verglichen werden. Bei einem Bauplan handelt es sich um eine Liste mit Merkmalen, die jedes Haus erhalten soll. Der Bauplan sagt »Außenanstrich«. Das aktuelle Haus-Objekt hat einen grauen Außenanstrich. Der Bauplan sagt »Küchenschrank«. Das aktuelle Haus-Objekt hat einen Küchenschrank im Stil Ludwig XIV.

Die Analogie hört nicht mit einer Liste der Merkmale auf. Zwischen Bauplänen und Klassen gibt es eine weitere wichtige Parallele. Ein Jahr, nachdem Sie den Bauplan erstellt haben, verwenden Sie ihn, um zehn weitere Häuser zu bauen. Und so funktioniert das auch mit Klassen und Objekten. Der Programmierer schreibt als Erstes Code, um eine Klasse zu beschreiben. Wenn das Programm dann läuft, erzeugt der Computer aus den Klassen (dem Bauplan) Objekte.

Und damit kennen Sie die wirkliche Beziehung zwischen Klassen und Objekten. Der Programmierer definiert eine Klasse, und der Computer macht aus den Klassen einzelne Objekte.

Was ist das Besondere an einer objektorientierten Sprache?

Stellen Sie sich anhand der Geschichte über den Hausbau aus dem vorherigen Abschnitt vor, dass Sie bereits ein Computerprogramm geschrieben haben, um die Bauanweisungen für neue Häuser zu kontrollieren. Und dann entscheidet der große Boss, den Plan zu ändern – in einen Plan, bei dem die Hälfte der Häuser drei und die andere Hälfte vier Schlafzimmer bekommt.

Wenn Sie im alten FORTRAN/C-Stil programmiert haben, sieht Ihre Anweisung so aus:

```
Graben Sie einen Graben für das Fundament.  
Schalen Sie die Seiten des Grabens ein, damit betoniert werden kann.  
Bringen Sie an den Seiten Kanthölzer an, die das Gerüst des Fundaments bilden.
```

...

Hier arbeiten Sie wie ein Architekt, der statt eines Bauplans eine lange Liste mit Anweisungen erstellt. Wenn Sie den Plan ändern wollen, müssen Sie die gesamte Liste abarbeiten, um die Anweisungen zu finden, die mit dem Bau von Schlafzimmern zu tun haben. Und damit das alles dann nicht zu einfach wird, könnten diese Anweisungen über die Seiten 234, 394 – 410, 739, 10 und 2 verteilt sein. Und wenn der Bauleiter auch noch die komplizierten Anweisungen einer dritten Person entziffern muss, dauert die Aufgabe zehnmal so lange.

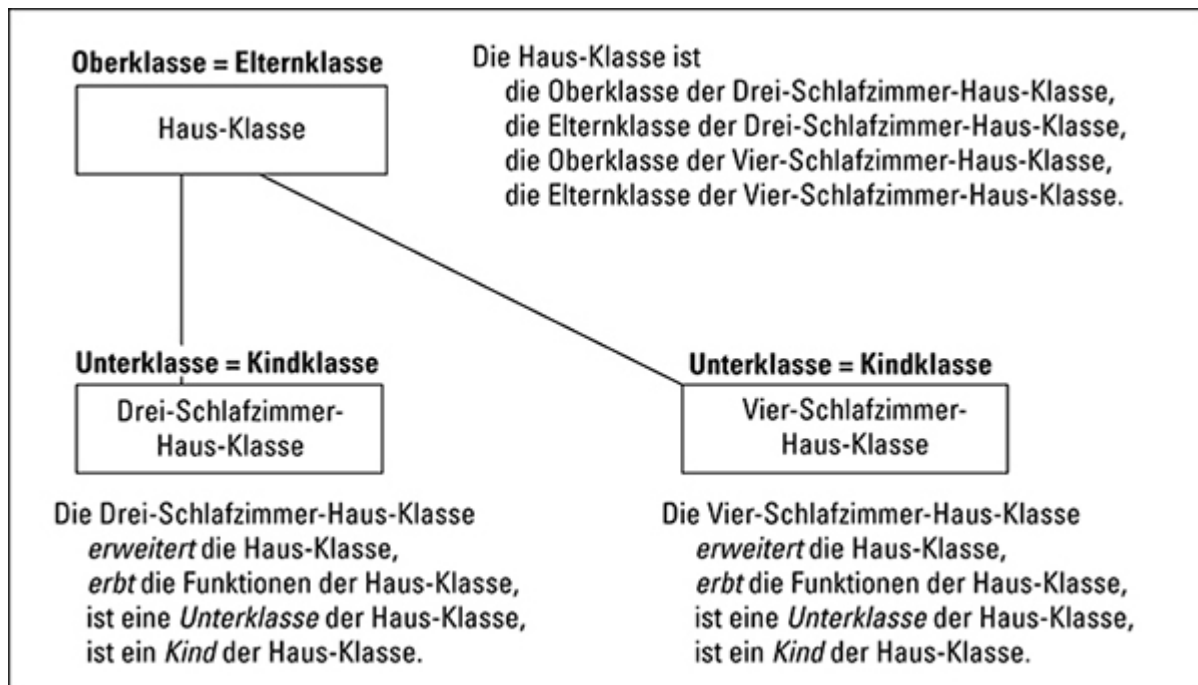
Wenn Sie nun aber mit einer Klasse beginnen, ist das so, als wenn Sie einen Bauplan anfangen. Wenn Sie festlegen, dass es sowohl Häuser mit drei als auch mit vier Schlafzimmern geben soll, beginnen Sie mit einem Bauplan, der *Haus* heißt und ein Erdgeschoss und einen ersten Stock enthält, wobei es in diesem ersten Stock noch keine Innenwände gibt. Dann erstellen Sie zwei Baupläne für den ersten Stock – einen für Häuser mit drei Schlafzimmern und einen für Häuser mit vier Schlafzimmern. Sie nennen diese Baupläne *Drei-Schlafzimmer-Haus* und *Vier-Schlafzimmer-Haus*.

Ihre Kollegen im Bauunternehmen sind über Ihre logische und strukturierte Vorgehensweise angenehm überrascht, aber sie haben Bedenken. Es gibt eine Frage: »Du hast einen der Baupläne *Drei-Schlafzimmer-Haus* genannt. Wieso das, wo es doch nur um ein Stockwerk und nicht um ein ganzes Haus geht?«

Sie lächeln wissend und antworten: »Der Plan mit den drei Schlafzimmern sagt, dass man sich für Informationen über das Erdgeschoss den ursprünglichen Bauplan ansehen soll. Auf diese Weise beschreibt der Drei-Schlafzimmer-Haus-Bauplan ein ganzes Haus. Dasselbe sagt der Vier-Schlafzimmer-Haus-Bauplan aus. Damit sind wir in der Lage, die ganze Arbeit zu nutzen, die wir bereits in den ursprünglichen Bauplan für ein Haus gesteckt haben, und können viel Geld sparen.«

In der Sprache der objektorientierten Programmierung *erben* die Klassen Drei- und Vier-Schlafzimmer-Haus die Funktionen der ursprünglichen Klasse Haus. Sie können auch sagen, dass die Klassen Drei- und Vier-Schlafzimmer-Haus die ursprüngliche Klasse Haus *erweitern* (siehe [Abbildung 1.1](#)).

Die ursprüngliche Klasse Haus wird die *Superklasse* der Klassen Drei- und Vier-Schlafzimmer-Haus genannt. Umgekehrt sind die Klassen Drei- und Vier-Schlafzimmer-Haus *Unterklassen* der ursprünglichen Klasse Haus. Doch damit nicht genug. Die ursprüngliche Klasse Haus wird auch *Elternklasse* der Klassen Drei- und Vier-Schlafzimmer-Haus genannt. Dementsprechend sind die Klassen Drei- und Vier-Schlafzimmer-Haus die *Kindklassen* der ursprünglichen Klasse Haus (siehe [Abbildung 1.1](#)).



[Abbildung 1.1](#): Die Terminologie der objektorientierten Programmierung

Es muss nicht ausdrücklich darauf hingewiesen werden, dass Ihre Kollegen im Bauunternehmen neidisch sind. Sie werden von vielen Kollegen umringt, die alles über Ihre neue Idee wissen wollen. Das ist genau der richtige Augenblick, um die nächste Bombe hochgehen zu lassen: »Indem wir Klassen mit Unterklassen erstellen, sind wir in der Lage, den Bauplan auch zukünftig wiederzuverwenden. Und wenn dann jemand kommt und ein Haus mit fünf Schlafzimmern haben will, erweitern wir unseren ursprünglichen Bauplan, indem wir einen Fünf-Schlafzimmer-Haus-Bauplan anlegen. Wir werden nie wieder Geld für einen originalen Bauplan für ein Haus ausgeben müssen.«

»Aber«, sagt einer der Kollegen in den hinteren Reihen, »was geschieht, wenn jemand ein anderes Erdgeschoss haben möchte? Wirfst du dann den ursprünglichen Bauplan des Hauses auf den Müll oder übermalst du den ursprünglichen Bauplan? Das wird ganz schön teuer, oder?«

Sie antworten in einem vertraulichen Ton: »Wir müssen am ursprünglichen Bauplan nichts ändern. Wenn jemand in seinem

Wohnzimmer einen Whirlpool haben möchte, erstellen wir einfach einen neuen kleinen Bauplan, der nur das neue Wohnzimmer beschreibt, und nennen ihn *Whirlpool-im-Wohnzimmer-Haus*-Bauplan. Dieser neue Bauplan kann auf den ursprünglichen Bauplan *Haus* verweisen, um auch weiterhin Zugriff auf die Informationen über den Rest des Hauses (den Teil, der sich außerhalb des Wohnzimmers befindet) zu haben.« In der Sprache der objektorientierten Programmierung *erweitert* der Bauplan *Whirlpool-im-Wohnzimmer-Haus* den ursprünglichen Haus-Bauplan. Der Whirlpool-Bauplan ist außerdem eine Unterklasse des ursprünglichen Haus-Bauplans. Die gesamte Terminologie wie Superklasse, Elternklasse und Kindklasse gilt auch hier. Neu ist nur, dass der Whirlpool-Bauplan die ursprünglichen Funktionen des Wohnzimmers *überschreibt*.

In den Tagen vor den objektorientierten Sprachen durchlebte die Programmierwelt eine Krise der Softwareentwicklung. Programmierer schrieben Code, dann entdeckten sie neue Bedürfnisse und mussten ihren Code auf den Müll werfen und ganz von vorn neu anfangen. Dies geschah immer wieder, weil der Code, den die Programmierer geschrieben haben, nicht wiederverwendet werden konnte. Dies alles wurde durch die objektorientierte Programmierung zum Besseren geändert (und wie Burd gesagt hat, ist Java »eine großartige objektorientierte Sprache«).

Ihr Verständnis für Klassen und Objekte vertiefen

Wenn Sie in Java programmieren, arbeiten Sie ständig mit Klassen und Objekten. Diese beiden Konzepte sind wirklich wichtig. Das ist auch der Grund dafür, warum ich Ihnen in diesem Kapitel eine Analogie von Klassen und Objekten nach der anderen um die Ohren haue.

Schließen Sie für eine Minute Ihre Augen und denken Sie darüber nach, was es bedeutet, ein Stuhl zu sein.

Ein Stuhl hat eine Sitzfläche, eine Rückenlehne und Beine. Jede Sitzfläche hat eine Form, eine Farbe, einen Härtegrad und so weiter. Dies sind die Eigenschaften eines Stuhls. In der objektorientierten Terminologie beschreibe ich die Klasse `Stuhl`.

Werfen Sie nun einen Blick über den Rand dieses Buches hinaus und schauen Sie sich einmal in Ihrem Zimmer um. (Wenn Sie gerade nicht in einem Zimmer sitzen, stellen Sie es sich vor.)

Im Zimmer gibt es mehrere Stühle, und jeder Stuhl ist ein Objekt. Jedes dieser Objekte ist ein Beispiel dieses ätherischen Elements, das als `Stuhl`-Klasse bezeichnet wird. Und genau so funktioniert das – die Klasse ist die Vorstellung von Bestuhlung, und jeder einzelne Stuhl ist ein Objekt.



Eine Klasse ist nicht wirklich eine Sammlung von Elementen. Stattdessen handelt es sich bei einer Klasse um die Idee hinter einer bestimmten Art von Elementen. Wenn ich über die Klasse Stühle in Ihrem Zimmer spreche, meine ich die Tatsache, dass jeder Stuhl Beine, eine Sitzfläche, eine Farbe und anderes hat. Die verschiedenen Stühle im Raum können eine unterschiedliche Farbe haben, aber das macht nichts. Wenn Sie über eine Klasse von Elementen sprechen, geht es um die Eigenschaften, die jedes dieser Elemente besitzt.

Es macht Sinn, sich ein Objekt so vorzustellen, als ob es die greifbare Instanz einer Klasse sei. Und es ist wirklich so, dass die offizielle Terminologie mit dieser Denkweise übereinstimmt. Wenn Sie ein Java-Programm schreiben, in dem Sie eine Klasse `Stuhl` definieren, wird jeder reale Stuhl (der Stuhl, auf dem Sie gerade sitzen, der leere Stuhl direkt neben Ihnen und so weiter) als *Instanz* der Klasse `Stuhl` bezeichnet.

Hier kommt ein weiterer Weg, um sich ein Bild von einer Klasse zu machen. Stellen Sie sich eine Tabelle vor, die Ihre drei Bankkonten wiedergibt (siehe [Tabelle 1.1](#)).

Kontonummer	Kontoart	Kontostand
1613154228647	Giro	174,87

Kontonummer	Kontoart	Kontostand
1011123421220000	Kredit	−471,03
1617238133447	Sparen	247,38

Tabelle 1.1: Eine Tabelle mit Bankkonten

Stellen Sie sich die Spaltenüberschriften als eine Klasse vor, und denken Sie an die einzelnen Zeilen der Tabelle als Objekte. Die Spaltenüberschriften der Tabelle beschreiben die Klasse `Konto`.

Jedes Konto verfügt entsprechend der Spaltenüberschriften der Tabelle über eine Kontonummer, eine Kontoart und einen Kontostand. Wenn wir das in der Terminologie der objektorientierten Programmierung neu formulieren, hat jedes Objekt in der Klasse `Konto` (das heißt jede Instanz der Klasse `Konto`) eine Kontonummer, eine Kontoart und einen Kontostand. Daraus folgt, dass es sich bei der untersten Zeile der Tabelle um ein Objekt mit der Kontonummer *1617238133447* handelt. Dasselbe Objekt besitzt die Kontoart *Sparen* und einen Kontostand von *247,38*. Wenn Sie ein neues Konto eröffneten, hätten Sie ein weiteres Objekt, und die Tabelle würde um eine Zeile wachsen. Das neue Objekt wäre eine Instanz derselben Klasse `Konto`.

Wie geht es weiter?

Dieses Kapitel ist mit allgemeinen Beschreibungen von Elementen angefüllt. Beschreibungen dieser Art sind immer dann nützlich, wenn Sie mit etwas anfangen. Sie verstehen einige Dinge erst dann, wenn Sie Einzelheiten kennen. Und genau darum kümmern sich die nächsten Kapitel.

Blättern Sie also um. Das nächste Kapitel möchte nicht länger darauf warten, von Ihnen gelesen zu werden.

2

Alles über Software

In diesem Kapitel

- ▶ Die Rolle der Software Development Tools verstehen
 - ▶ Die für Sie richtige Java-Version auswählen
 - ▶ Vorbereitungen treffen, um Java-Programme zu schreiben und auszuführen
-

Die beste Art, um Java kennenzulernen, ist, mit Java zu arbeiten. Wenn Sie sich mit Java beschäftigen, schreiben und testen Sie Ihre eigenen Java-Programme und führen sie aus. Dieses Kapitel sorgt dafür, dass Sie mit Java arbeiten können, indem es die Einrichtung der Software beschreibt, die es auf Ihrem Computer geben muss, und zwar unabhängig davon, ob dieser unter Windows, Mac, Linux oder Ottos selbst gestricktem Betriebssystem läuft. Dieses Kapitel beschreibt *keine* Installationsabläufe, die speziell für Windows, für einen Mac oder für ein anderes Betriebssystem gelten.

Anleitungen für einen Schnellstart

Wenn Sie ein erfahrener Computerveteran sind und sich im Computerwesen auskennen (was auch immer das heißen mag) oder wenn Sie keine Lust haben, sich mit langen Installationsanleitungen abzugeben, können Sie die benötigte Software installieren, indem Sie den allgemeinen Anleitungen folgen, die dieses Kapitel für Sie bereithält. Diese Anleitungen funktionieren bei vielen, wenn auch nicht bei allen Computern. Und Sie finden in diesem Abschnitt auch keine in die Einzelheiten

gehende Schritte, keine Wenn-dies-dann-das-Alternativen und keine Das-funktioniert-zwar-aber-das-wäre-besser-Tipps.

Um Ihren Computer auf das Schreiben von Java-Programmen vorzubereiten, gehen Sie so vor:

✓ **Besuchen Sie Java.com.**

Folgen Sie den Anweisungen unter java.com, um Java herunterzuladen und zu installieren. Achten Sie darauf, die für Ihr Betriebssystem passende Version (Windows, Mac, Linux, 32-Bit, 64-Bit) auszuwählen. Dies muss gegebenenfalls manuell erfolgen, weil die Erkennungsautomatik nicht immer sauber funktioniert.



Unter de.wikibooks.org/wiki/Java finden Sie deutschsprachige Anleitungen für den Umgang mit Java – unter anderem auch eine ausführliche Installationsanleitung für Java.

✓ **Besuchen Sie optional**

www.oracle.com/technetwork/java/javase/downloads.

Folgen Sie den Anweisungen dieser Website, um die Java-SE-Dokumentation herunterzuladen und zu installieren. (Diese Anweisungen sind auch als *Javadoc Pages* oder *Java SE API Docs* bekannt.)

✓ **Besuchen Sie Eclipse.org.**

Folgen Sie den Anweisungen unter eclipse.org/downloads, um Eclipse herunterzuladen und zu installieren. Beachten Sie, dass Eclipse nur in Englisch zur Verfügung steht. Achten Sie darauf, die Version auszuwählen, die der installierten Java-Version entspricht.



Die Download-Seite von Eclipse bietet mehrere Pakete an, zu denen auch Eclipse Classic, Eclipse for Java EE, Eclipse for JavaScript und andere gehören. Um die Beispiele in diesem Buch ausführen zu können, benötigen Sie nur ein relativ kleines Eclipse-Paket – die Eclipse IDE for Java Developers.

Die Installation von Eclipse sieht so aus, dass Sie die heruntergeladene Datei nur entpacken und das entpackte Paket an einen Ort Ihrer Festplatte verschieben, an dem Sie es wiederfinden. Eclipse wird aus diesem Verzeichnis heraus über die Datei [eclipse.exe](#) gestartet.

✓ Testen Sie die installierte Software.

- Starten Sie Eclipse.
- Legen Sie in Eclipse ein neues Java-Projekt an.
- Erstellen Sie im Java-Projekt eine neue Java-Klasse und geben Sie ihr den Namen `Displayer`.
- Bearbeiten Sie die neue Datei `Displayer.java`, indem Sie den Code in [Listing 3.1](#) (das erste Codelisting in [Kapitel 3](#)) eingeben. Geben Sie diesen Code im Eclipse-Fensterelement EDITOR ein.
- Führen Sie `Displayer.java` aus und überprüfen Sie das Ergebnis des Laufs, das Sie werden `Java lieben!` sein sollte.

Das wär's dann. Aber denken Sie daran, dass nicht jeder (Computerfreak oder nicht) diesem Anweisungsskelett fehlerfrei folgen kann. Aus diesem Grund gibt es Hilfsmittel:

✓ Besuchen Sie die amerikanische Website dieses Buches, wenn Sie über (halbwegs) gute Englischkenntnisse verfügen.

Gehen Sie nicht über Los. Probieren Sie die Schnellstartanleitungen dieses Kapitels nicht aus. Folgen Sie den detaillierteren, nur in englischer Sprache vorliegenden Anleitungen, die Sie unter

www.allmycode.com/JavaForDummies finden.

✓ **Probieren Sie die Schnellstartanleitungen in diesem Abschnitt aus.**

Sie können beim Ausprobieren niemanden verletzen. Wenn Sie unglücklicherweise die falsche Software installiert haben, können Sie diese zur Not auf Ihrem Computer belassen. (Sie müssen sie nicht wieder deinstallieren.) Wenn Sie sich nicht sicher sind, ob Sie die Software richtig installiert haben, können Sie sich immer noch mit den detaillierten Anleitungen auf meiner Website beschäftigen.

✓ **Schicken Sie entweder Ihre Fragen per E-Mail (bitte nur in englischer Sprache) an**

JavaForDummies@allmycode.com oder besuchen Sie die Webseite des deutschen Buches unter www.wiley-vch.de/publish/dt/books/ISBN3-527-71070-1.

✓ **Twittern Sie unter @allmycode mit mir.**

✓ **Besuchen Sie meine Facebook-Seite: /allmycode.**

Ich liebe es, von Lesern zu hören.

Was Sie auf Ihrem Computer installieren

Ich habe einmal einen Werkzeugmacher und Formenbauer getroffen. Er verwendete Werkzeuge, um Werkzeuge (und Formen) herzustellen. Ich war froh, ihn kennengelernt zu haben, weil ich wusste, dass ich irgendwann einmal einen Vergleich von Computerprogrammen und dem Herstellen von Werkzeugen benötigen würde.

Ein Computerprogramm verwendet vorhandene Programme als Werkzeuge, um neue Programme zu erstellen. Die vorhandenen Programme und die neuen Programme können ganz unterschiedliche Aufgaben ausführen. So kann zum Beispiel ein Java-Programm (ein Programm, das Sie erstellt haben) den Überblick über Geschäftskunden behalten. Damit dieses

Programm erstellt werden konnte, haben Sie vielleicht ein vorhandenes Programm verwendet, das sich um Fehler in Ihrem Java-Code gekümmert hat. Dieses Programm, das nur der Fehlersuche dient, ist in der Lage, Fehler in jeder Art von Java-Code zu finden – Code, um einen Überblick über Kunden zu bekommen, Code für die Wettervorhersage oder Code für Apps auf Handys.

Wie viele Werkzeuge benötigen Sie nun für Ihre Java-Programme? Als Neuling kommen Sie mit drei aus:

✓ **Sie benötigen einen *Compiler*.**

Ein Compiler (der im Deutschen auch *Kompilator* oder *Kompilierer* genannt wird) nimmt den Java-Code, den Sie geschrieben haben, und wandelt diesen Code in etwas um, das auf Ihrem Computer ablaufen kann.

✓ **Sie benötigen eine *Java Virtual Machine (JVM)*.**

Eine Java Virtual Machine führt Ihren Code (und den Java-Code anderer Leute) auf Ihrem Computer aus.

✓ **Sie benötigen eine *integrierte Entwicklungsumgebung (IDE)*.**

Eine integrierte Entwicklungsumgebung (englisch *Integrated Development Environment* oder *IDE*) hilft dabei, Ihren Code zu verwalten, und stellt Ihnen bequeme Möglichkeiten zur Verfügung, um Code zu schreiben, zu kompilieren und ablaufen zu lassen.

Im World Wide Web gibt es von jedem dieser Werkzeuge kostenlos herunterladbare Versionen. So raten Ihnen zum Beispiel die Schnellstartanleitungen am Anfang dieses Kapitels, *Java.com* und *Eclipse.org* zu besuchen. Wenn Sie auf der Seite von *Java.com* auf eine Schaltfläche klicken, installieren Sie auf Ihrem Computer eine Java Virtual Machine. Auf *Eclipse.org* laden Sie das Eclipse Integrated Development Environment herunter, das einen eigenen, eingebauten Java-Compiler enthält. (Sie erhalten damit zwei der drei benötigten Komponenten in einem Download. Nicht übel, oder?)

Der Rest dieses Kapitels beschreibt Compiler, JVMs und IDEs.

Was ist ein Compiler?

»Ein Compiler nimmt sich den Java-Code, den Sie schreiben, und wandelt diesen Code in etwas um, das auf Ihrem Computer ausgeführt werden kann.«

Barry Burd, Java für Dummies

Sie sind ein menschliches Wesen. (Okay, jede Regel kennt Ausnahmen, aber wenn Sie dieses Buch lesen, sind Sie höchstwahrscheinlich menschlich.) Und Menschen sind in der Lage, den Code in [Listing 2.1](#) zu schreiben und zu verstehen.

```
// Dies ist ein Teil eines Java-Programms
// (kein vollständiges Java-Programm).
roomNum = 1;
while (roomNum < 100) {
    if (guests[roomNum] == 0) {
        out.println("Room " + roomNum
            + " is available.");
        exit(0);
    } else {
        roomNum++;
    }
}
out.println("No vacancy");
```

[Listing 2.1](#): Nach einem freien Zimmer suchen

Der Code in [Listing 2.1](#) prüft, ob es in einem kleinen Hotel (mit den Zimmernummern 1 bis 99) freie Zimmer gibt. Sie könnten diesen Code ablaufen lassen, ohne weitere Zeilen hinzufügen zu müssen. Wichtig ist hier, dass Sie hinter die ungewöhnliche Interpunktion des Codes kommen, damit Sie erkennen, was der Code versucht zu machen:

```
Setze die Zimmernummer auf 1.
Solange die Zimmernummer kleiner als 100 ist,
    prüfe die Anzahl der Gäste im Zimmer.
    Wenn die Zahl der Gäste im Zimmer 0 ist, dann
        gib aus, dass das Zimmer verfügbar ist,
        und beende das Programm.
Anderenfalls,
```

sei bereit, das nächste Zimmer zu überprüfen,
indem die Zimmernummer um 1 erhöht wird.
Wenn die nicht existierende Zimmernummer 100 erscheint,
dann gib aus, dass alle Zimmer belegt sind.

Wenn Sie die Ähnlichkeit zwischen [Listing 2.1](#) und seinem deutschen Äquivalent nicht erkennen, sollten Sie sich nichts daraus machen. Sie lesen gerade *Java für Dummies* und können wie die meisten menschlichen Wesen lernen, den Code in [Listing 2.1](#) zu schreiben und zu lesen. Code dieser Art wird auch *Java-Quellcode* genannt.

Und das ist der Haken an der Sache: Computer sind keine Menschen. Normalerweise folgen Computer keinen Anweisungen wie denen in Listing 2.1. Oder kurz gesagt, Computer folgen keinen Anweisungen, die aus Java-Quellcode bestehen. Computer folgen stattdessen kryptischen Befehlen wie denen in Listing 2.2.

```
aload_0
iconst_1
putfield Hotel/roomNum I
goto 32
aload_0
getfield Hotel/guests [I
aload_0
getfield Hotel/roomNum I
iaload
ifne 26
getstatic java/lang/System/out Ljava/io/PrintStream;
new java/lang/StringBuilder
dup
ldc "Room "
invokespecial java/lang/StringBuilder/<init>
(Ljava/lang/String;)V
aload_0
getfield Hotel/roomNum I
invokevirtual java/lang/StringBuilder/append(I)Ljava/
lang/StringBuilder;
ldc " is available."
invokevirtual
```

```

    java/lang/StringBuilder/append(Ljava/lang/String;)L
java/lang/
StringBuilder;
invokevirtual java/lang/StringBuilder/toString()Ljava
/lang/String;
invokevirtual java/io/PrintStream/println(Ljava/lang/
String;)V
iconst_0
invokestatic java/lang/System/exit(I)V
goto 32
aload_0
dup
getfield Hotel/roomNum I
iconst_1
iadd
putfield Hotel/roomNum I
aload_0
getfield Hotel/roomNum I
bipush 100
if_icmplt 5
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "No vacancy"
invokevirtual java/io/PrintStream/println(Ljava/lang/
String;)V
return

```

[Listing 2.2](#): Die Anweisungen in [Listing 2.1](#) – übersetzt in Java-Bytecode

Bei den Anweisungen in [Listing 2.2](#) handelt es sich nicht um Java-Quellcode. Es sind Anweisungen in Form von *Java-Bytecode*. Wenn Sie ein Java-Programm schreiben, schreiben Sie Quellcode-Anweisungen (wie die in Listing 2.1). Nachdem Sie den Quellcode geschrieben haben, führen Sie auf Ihren Quellcode ein Programm aus (das heißt, Sie wenden ein Werkzeug an). Bei diesem Programm handelt es sich um einen *Compiler*. Der Compiler übersetzt Ihre Quellcode-Anweisungen in Java-Bytecode-Anweisungen. Oder mit anderen Worten, der Compiler nimmt Code, den Sie schreiben und verstehen (wie den in Listing 2.1), und übersetzt ihn in Code, den ein Computer ausführen kann (wie den in Listing 2.2).



Sie könnten Ihren Quellcode in eine Datei mit dem Namen `Hotel.java` packen. In diesem Fall würde der Compiler den Java-Bytecode in einer Datei mit dem Namen `Hotel.class` ablegen. Normalerweise müssen Sie sich nicht mit dem Bytecode in der Datei `Hotel.class` abgeben. Es ist nämlich so, dass der Compiler die Datei `Hotel.class` nicht wie eine normale Textdatei kodiert, was dazu führt, dass Sie den Bytecode nicht mit einem normalen Texteditor untersuchen können. Wenn Sie versuchen, `Hotel.class` mit Notepad, TextEdit, KWrite oder Microsoft Word zu öffnen, sehen Sie nichts als Punkte, Gekrakel und anderes Kauderwelsch. Um [Listing 2.2](#) zu erstellen, musste ich für meine Datei `Hotel.class` ein besonderes Werkzeug einsetzen. Dieses Werkzeug zeigt eine textähnliche Version der Java-Bytecodedatei an. Ich habe Ando Saabas's Java Bytecode Editor (www.cs.ioc.ee/~ando/jbe) verwendet.



Niemand (außer vielleicht ein paar verrückte Entwickler, die an sehr entlegenen Orten ein einsames Leben führen) schreibt Java-Bytecode. Sie führen Software (einen Compiler) aus, um Java-Bytecode zu erstellen. Der einzige Grund, sich mit [Listing 2.2](#) zu beschäftigen, ist zu verstehen, wie hart Ihr Computer arbeiten muss.

Was ist eine Java Virtual Machine?

»Eine Java Virtual Machine führt auf Ihrem Computer Ihren Java-Code und den anderer Leute aus.«

Barry Burd, Java für Dummies

Ich habe im letzten Abschnitt *Was ist ein Compiler?* viel Wind um Computer gemacht, die Anweisungen wie denen in [Listing 2.2](#) folgen. Wenn sich die Aufregung erst einmal gelegt hat, werden Sie feststellen, dass vielleicht doch etwas hängen geblieben ist. Wenn Sie die-sen Abschnitt aber nur überflogen haben, ist Ihnen vielleicht etwas entgangen. Dort heißt es: »... Computer folgen stattdessen kryptischen Anweisungen wie denen in Listing 2.2.«

Die Anweisungen in [Listing 2.2](#) haben viel mit Anweisungen (Befehlen) gemeinsam, die ein Computer ausführen kann, aber allgemein gilt, dass Computer nichts direkt mit Java-Bytecode-Anweisungen anfangen können. Stattdessen verfügt jeder Computer-Prozessor über einen eigenen Satz ausführbarer Befehle, und jedes Betriebssystem verwendet den Befehlssatz eines Prozessors auf etwas andere Weise.

Hier eine hypothetische Situation: Stellen Sie sich vor, dass Sie das Betriebssystem Linux auf einem Computer mit einem alten Pentium-Prozessor einsetzen. Ein Freund lässt Linux auf einem Computer ablaufen, der einen etwas anderen Prozessor besitzt – einen PowerPC-Prozessor. (In den 1990ern stellt Intel Pentium- und IBM PowerPC-Prozessoren her.)

[Listing 2.3](#) enthält einen Satz mit Anweisungen, die auf dem Bildschirm `Hello world!` ausgeben. (Ich habe diese Intel-Anweisungen dem Buch *Linux Assembly HOWTO* von Konstantin Boldyshevs entnommen – tldp.org/HOWTO/Assembly-HOWTO/hello.html.) Diese Anweisungen funktionieren auf einem Pentium-Prozessor, der Linux ausführt.

```
.data
msg:
    .ascii "Hello, world!\n"
    len = . - msg

.text
    .global _start
_start:
    movl    $len,%edx
    movl    $msg,%ecx
    movl    $1,%ebx
    movl    $4,%eax
    int     $0x80
    movl    $0,%ebx
    movl    $1,%eax
    int     $0x80
```

[Listing 2.3](#): Ein einfaches Programm für einen Pentium-Prozessor

[Listing 2.4](#) enthält einen anderen Befehlssatz, der auf dem Bildschirm `Hello world!` ausgibt. (Ich habe den PowerPC-

Code Hollis Blanchards PowerPC-Assembly-Seite entnommen – www.ibm.com/developerworks/library/l-ppc.) Die Anweisungen in [Listing 2.4](#) funktionieren auf einem PowerPC-Prozessor, der mit Linux läuft.

```
.data
msg:
    .string "Hello, world!\n"
    len = . - msg
.text
    .global _start
_start:
    li      0,4
    li      3,1
    lis     4,msg@ha
    addi    4,4,msg@l
    li      5,len
    sc
    li      0,1
    li      3,1
    sc
```

[Listing 2.4](#): Ein einfaches Programm für einen PowerPC-Prozessor

Die Anweisungen in [Listing 2.3](#) laufen problemlos auf einem Pentium-Prozessor, aber sie sagen einem PowerPC-Prozessor nichts. Ähnlich ist das mit den Befehlen in Listing 2.4, mit denen zwar ein PowerPC-Prozessor viel, ein Pentium-Prozessor aber nichts anfangen kann. Aus diesem Grund steht die PowerPC-Software Ihres Freundes nicht für Ihren Computer zur Verfügung. Und die Software Ihres Intel-Computers wird nicht unbedingt auch auf dem Computer Ihres Freundes laufen können.

Besuchen Sie nun eine Cousine. Deren Computer besitzt einen Pentium-Prozessor (wie Ihr Computer), aber auf dem Computer Ihrer Cousine läuft Windows und nicht Linux. Was macht der Computer Ihrer Cousine, wenn Sie ihn mit dem Pentium-Code in [Listing 2.3](#) füttern? Er schreit empört: »Das ist keine gültige Win32-Anwendung!« oder »Windows kann diese Datei nicht öffnen.« Was für ein Mist!

Java-Bytecode bringt Ordnung in dieses Chaos. Java-Bytecode ist so etwas wie der Code in den [Listing 2.3](#) und 2.4, aber Java-Bytecode ist nicht auf einen Prozessortyp oder ein Betriebssystem fokussiert. Stattdessen laufen Java-Bytecode-Anweisungen auf jedem Computer. Wenn Sie ein Java-Programm schreiben und es zu Bytecode kompilieren, ist nicht nur Ihr Computer in der Lage, das Programm auszuführen, sondern auch der Computer Ihres Freundes kann den Bytecode ausführen, der Supercomputer Ihrer Großmutter kann den Bytecode ausführen, und wenn Sie ein wenig Glück haben, kann auch Ihr niedliches, kleines Handy den Bytecode ausführen.



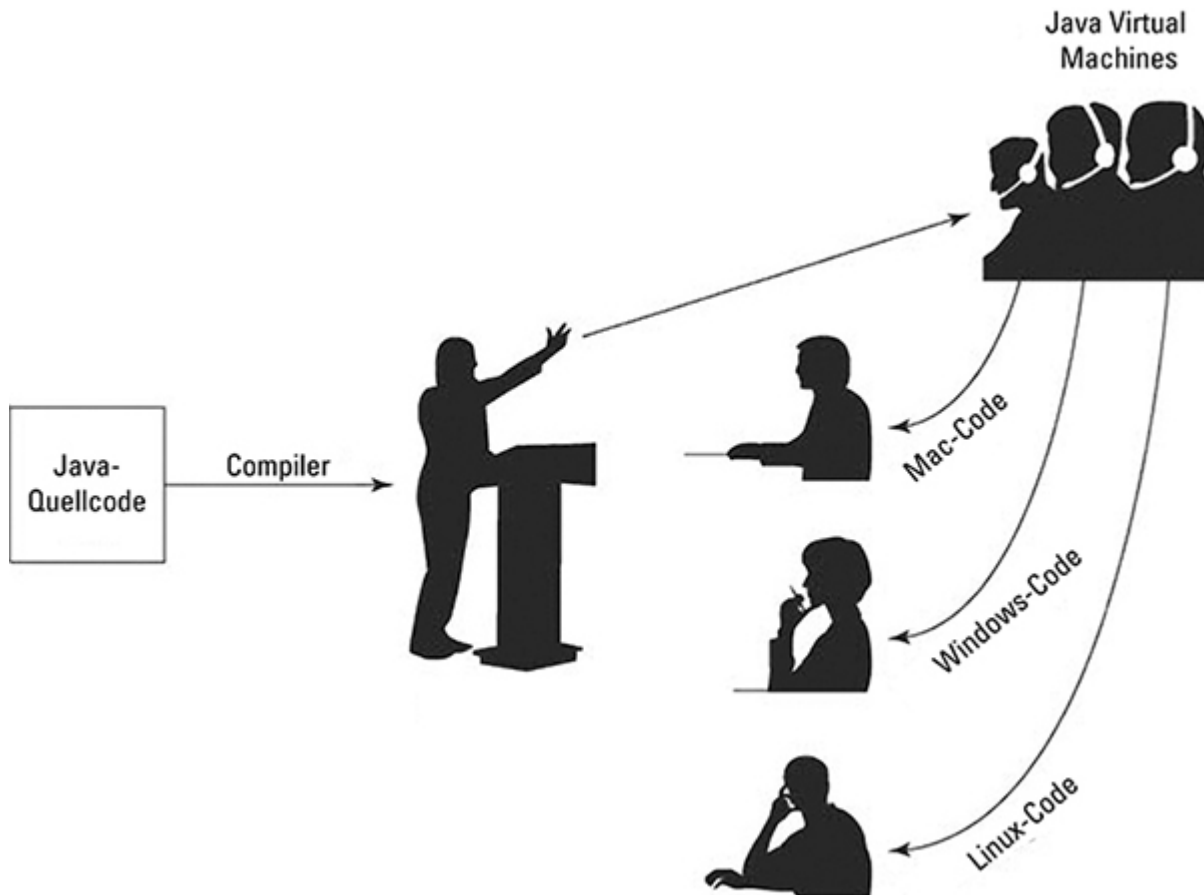
Wenn Sie Java-Bytecode sehen wollen, werfen Sie einen Blick auf [Listing 2.2](#). Denken Sie aber immer daran, dass Sie Java-Bytecode niemals schreiben oder entziffern müssen. Das Schreiben von Bytecode ist Aufgabe des Computers. Das Entziffern von Bytecode ist die Aufgabe der Java Virtual Machine.

Bei Java können Sie eine Bytecodedatei nehmen, die Sie mit einem Windows-Computer erstellt haben, den Bytecode auf einen x-beliebigen Computer kopieren und den Code problemlos ausführen. Dies ist einer der vielen Gründe, warum Java so schnell so beliebt geworden ist. Diese außergewöhnliche Fähigkeit, die es Ihnen erlaubt, Code auf vielen verschiedenen Arten von Computern ablaufen zu lassen, wird *Portierbarkeit* genannt.

Was nun macht Java-Bytecode so vielseitig? Diese unglaubliche universelle Einsetzbarkeit des Java-Bytecodes hat ihren Ursprung in der Java Virtual Machine. Die Java Virtual Machine ist eines der drei Werkzeuge, die es auf Ihrem Computer geben muss.

Stellen Sie sich vor, dass Sie der Windows-Repräsentant beim United Nations Security Council sind (siehe [Abbildung 2.1](#)). Rechts neben Ihnen sitzt der Vertreter von Macintosh und links von Ihnen der Linux-Repräsentant. (Natürlich sind Sie mit diesen Leuten nicht dick befreundet, auch wenn Sie ganz gut miteinander

auskommen. Was haben Sie erwartet? Das nennt man Politik!) Am Rednerpult steht der hervorragende Vertreter von Java. Er spricht Bytecode-isch und weder Sie noch Ihre Kollegen (Mac und Linux) verstehen auch nur ein Wort Java-Bytecode.



[Abbildung 2.1](#): Ein imaginäres Treffen des UN Security Councils

Aber jeder von Ihnen hat einen *Interpreter*. Ihr Interpreter übersetzt Bytecode in die Windows-Sprache, während der Java-Repräsentant redet. Ein anderer Interpreter übersetzt Bytecode in Macintosh-isch. Und ein dritter Interpreter übersetzt Bytecode in die Linux-Sprache.

Stellen Sie sich Ihren Interpreter als einen virtuellen Botschafter vor. Der Interpreter repräsentiert Ihr Land nicht wirklich, aber er führt eine der wichtigen Aufgaben aus, die auch ein echter Botschafter erledigt. Der Interpreter hört in Ihrem Auftrag dem

Bytecode zu. Er macht dann das, was auch Sie machen würden, wenn Ihre Landessprache Java-Bytecode wäre. Der Interpreter gibt vor, der Windows-Botschafter zu sein, tut sich die langweilige Bytecode-Rede an, nimmt jedes Wort auf und verarbeitet jedes Wort auf irgendeine Weise.

Sie besitzen einen Interpreter – einen virtuellen Botschafter. Und auf dieselbe Weise führt ein Windows-Computer seine eigene Bytecode interpretierende Software aus. Diese Software ist die Java Virtual Machine.

Eine Java Virtual Machine (JVM) ist ein Stellvertreter, ein Botenjunge, ein Vermittler. Die JVM dient als Dolmetscher zwischen Javas Ich-laufe-überall-Bytecode und dem System Ihres Computers. Die JVM führt Ihren Computer, während er läuft, durch die Anweisungen des Bytecodes. Sie untersucht Ihren Bytecode Bit für Bit und holt die Anweisungen heraus, die im Bytecode beschrieben werden. Die JVM übersetzt Bytecode für Ihre Windows-, Mac- oder Linux-Systeme oder welche Art von Computer Sie benutzen. Das ist eine gute Sache. Und das ist genau das, was Java-Programme portierbarer macht als Programme, die in irgendeiner anderen Sprache geschrieben worden sind.

Was um Himmels willen ist Java 2 Standard Edition 1.2?

Wenn Sie ein wenig im Web herumstöbern und nach Java-Werkzeugen suchen, finden Sie Dinge, die alle möglichen komischen Namen tragen. Sie finden das Java Development Kit, das Software Development Kit, die Java Runtime Environment und andere verwirrende Bezeichnungen.

- ✓ Die Namen *Java Development Kit* (JDK) und *Software Development Kit* (SDK) bezeichnen unterschiedliche Versionen desselben Satzes an Werkzeugen – einen Werkzeugsatz, dessen Schlüsselkomponente ein Java-Compiler ist.
- ✓ Der Name *Java Runtime Environment* (JRE) bezeichnet einen Werkzeugsatz, dessen Schlüsselkomponente eine

Java Virtual Machine ist.

Wenn Sie das JDK auf Ihrem Computer installieren, enthält es bereits die JRE. Sie können die JRE aber auch als eigenständiges Programm bekommen. Es kommt sogar vor (und zwar gerade bei Entwicklern), dass viele Kombinationen von JDK und JRE auf einem Computer existieren. So gibt es zum Beispiel im Moment auf meinem Windows-Computer im Verzeichnis `C:\Programme\Java` das JDK 1.6, das JDK 1.8 und die JRE 8 und im Verzeichnis `C:\Programme (x86)\Java` die JRE 7. Und zwischen diesen Versionen kommt es nur selten zu Konflikten. Wenn Sie vermuten, dass es bei Ihnen einen Versionskonflikt geben könnte, sollten Sie am besten alle JDK- und JRE-Versionen mit Ausnahme der jüngsten Version (zum Beispiel JDK 1.8 oder JRE 8) deinstallieren.

Die Nummerierung der Java-Versionen ist verwirrend. Statt »Java 1«, »Java 2« und »Java 3« winden sich die Java-Versionen über einen Kurs voller Hindernisse.

Dieser Kasten hier beschreibt die Entwicklung neuer Java-Versionen über die Jahre hinweg. Jede Java-Version hat mehrere Namen. Bei der *Produktversion* handelt es sich um den offiziellen, allgemein gültigen Namen, und die *Entwicklungsversion* ist eine Zahl, die Versionen identifiziert, damit sie die Entwickler nachhalten können. (In ihren zwanglosen Unterhaltungen benutzen Entwickler alle möglichen Namen für die verschiedenen Java-Versionen.) Beim *Codenamen* handelt es sich um einen scherzhaften Namen, der eine Version identifiziert, während sie entwickelt wird.

Plattform	Codename	Funktionen

1995 (Beta)		

1996 JDK* 1,0		

1997 JDK 1.1		Innere Klassen, Java-Beans, Reflections

Plattform	Codename	Funktionen
1998 J2SE* 1.2	Playground	Swing-Klassen für das Erstellen von GUI-Interfaces
2000 J2SE 1.3	Kestrel	Java Naming and Directory Interface (JNDI)
2002 J2SE 1.4	Merlin	Neue I/O, reguläre Ausdrücke, XML kann geparkt werden
2004 J2SE 5.0	Tiger	Generische Typen, Annotations, enum-Typen, <code>varargs</code> , erweiterter <code>for</code> -Befehl, statischer Import, neue Nebenläufigkeitsklassen
2006 Java SE* 6	Mustang	Scripting-Sprachunterstützung, Verbesserungen beim Leistungsverhalten
2011 Java SE 7	Dolphin	Strings in <code>switch</code> -Befehlen, Fangen von Mehrfachausnahmen, <code>try</code> -Befehle mit Ressourcen, Zusammengehen mit JavaFX
2014 Java SE 8		Lambda-Ausdrücke

Die Sternchen in der Tabelle kennzeichnen Änderungen bei der Formulierung der Namen der Java-Produktversionen. 1996 hießen die Produktversionen *Java Development Kit 1.0* und *Java Development Kit 1.1*. 1998 entschied jemand, das Produkt *Java 2 Standard Edition 1.2* zu taufen, was bis heute jeden verwirrt. Damals wurde jeder, der den Begriff *Java Development Kit* verwendet hat, aufgefordert, den Ausdruck *Software Development Kit* (SDK) zu benutzen.

2004 verschwand das *1.* aus den Versionsnamen der Plattformen und 2006 verloren die Java-Plattformnamen auch die *2* und die *0*.

Zu den wohl bemerkenswertesten Änderungen für Java-Entwickler kam es 2004. Mit der Freigabe von J2SE 5.0 nahmen die Java-Aufpasser Änderungen

an der Sprache vor, indem neue Funktionen hinzugefügt wurden – Funktionen wie generische Typen, Annotations, `varargs` und den erweiterten `for`-Befehl.

(Wenn Sie Annotations in Aktion sehen wollen, sollten Sie zu den [Kapiteln 8](#) und [9](#) gehen. Beispiele, die die Verwendung von generischen Typen zeigen, und die erweiterte `for`-Anweisung finden Sie in [Kapitel 11](#).)

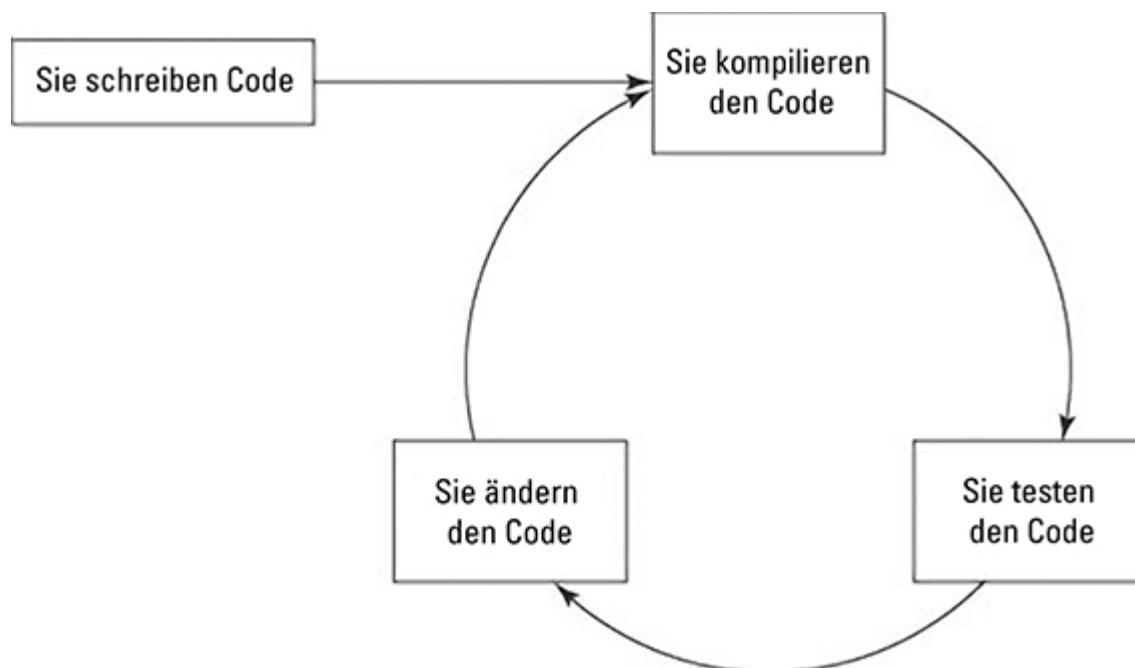
Die meisten Programme in diesem Buch laufen nur mit Java 5.0 oder höher ... Sie laufen nicht mit Java-Versionen vor 5.0. Auf keinen Fall kommen sie mit Java 1.4 oder Java 1.4.2 klar.

Software entwickeln

»Dies alles ist früher schon geschehen und wird auch wieder geschehen.«

Peter Pan (J. M. Barrie) und Battlestar Galactica (2003 – 2009, NBC)

Wenn Sie ein Java-Programm erstellen, wiederholen Sie immer wieder dieselben Schritte, [Abbildung 2.2](#) stellt diesen Kreislauf dar.



[Abbildung 2.2](#): Ein Java-Programm entwickeln

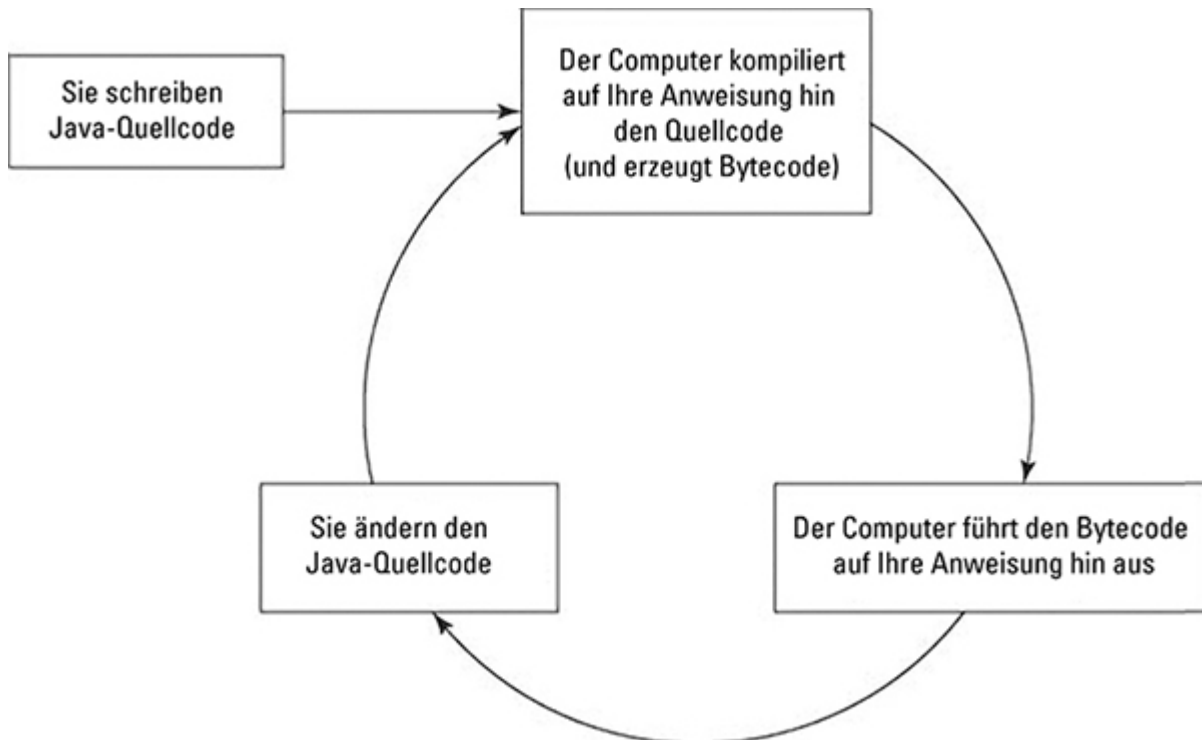
Als Erstes schreiben Sie ein Programm. Nachdem Sie den ersten Entwurf geschrieben haben, kompilieren Sie das Programm, führen es aus und ändern es immer wieder. Mit ein wenig Erfahrung verlieren die Schritte Kompilieren und Ausführen sehr schnell ihren Schrecken. In vielen Fällen reicht ein Mausklick aus, um das Kompilieren oder die Programmausführung zu starten.

Natürlich sind das Schreiben eines ersten Entwurfs und das Ändern des Codes keine Aufgaben, die sich mit einem Klicken erledigen lassen. Das Entwickeln von Code kostet Zeit und verlangt Konzentration.



Lassen Sie sich nicht entmutigen, wenn der erste Entwurf Ihres Codes nicht funktioniert. Lassen Sie sich auch nicht dadurch entmutigen, dass der fünfundzwanzigste Entwurf Ihres Codes nicht funktioniert. Das Neuschreiben von Code ist eines der wichtigsten Dinge, die Sie (neben der Sicherstellung des Weltfriedens) machen können.

Wenn über das Schreiben von Programmen gesprochen wird, werden häufig die Formulierungen aus [Abbildung 2.2](#) verwendet. Sie besagen: »Sie kompilieren den Code« und »Sie führen den Code aus.« Aber das »Sie« ist nicht immer zutreffend, und der »Code« ist bei jedem Durchlaufen des Kreislaufs ein wenig anders. [Abbildung 2.3](#) beschreibt den Kreislauf aus [Abbildung 2.2](#) detaillierter.



[Abbildung 2.3](#): Wer macht was mit welchem Code?



[Abbildung 2.3](#) enthält für viele viel zu viele Informationen. Wenn ich auf eine Schaltfläche AUSFÜHREN oder RUN klicke, muss ich mich nicht daran erinnern, dass der Computer für mich Code ausführt. Und meinetwegen kann der Computer meinen originalen Java-Code oder ein Bytecode-Imitat meines ursprünglichen Java-Codes ablaufen lassen. Die Einzelheiten in [Abbildung 2.3](#) sind nicht so wichtig. Der einzige Grund für [Abbildung 2.3](#) liegt darin, Ihnen zu helfen, falls Sie die ungenauen Bezeichnungen in [Abbildung 2.2](#) verwirren. Wenn dies nicht der Fall ist, ignorieren Sie [Abbildung 2.3](#).

Was ist eine integrierte Entwicklungsumgebung?

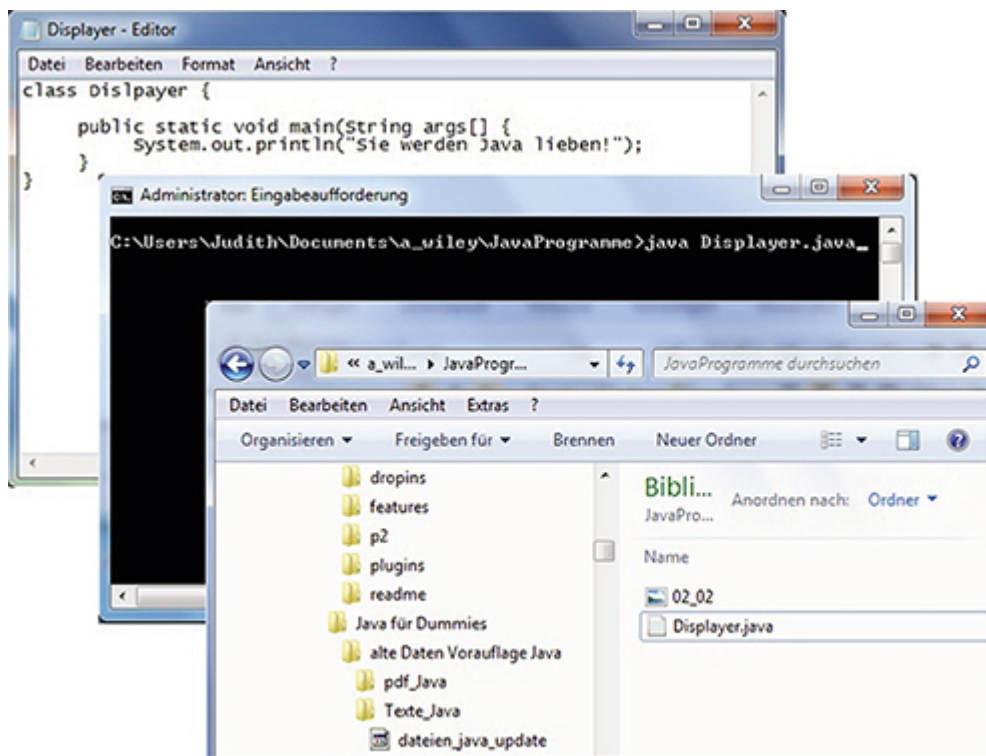
»Eine integrierte Entwicklungsumgebung hilft Ihnen dabei, Ihren Java-Code zu verwalten, und sorgt für die passenden

Wege, damit Sie Ihren Code schreiben, kompilieren und ausführen können.«

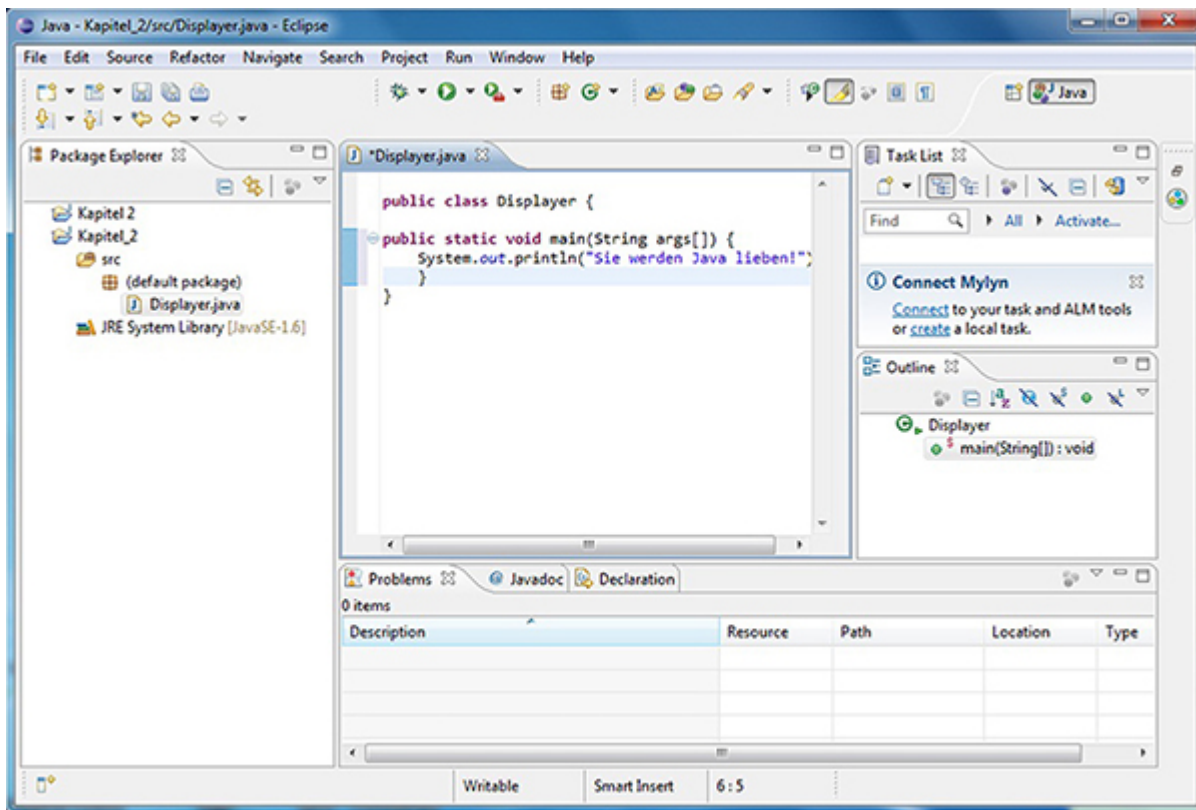
Barry Burd, Java für Dummies

Früher gehörte zum Schreiben und Ausführen eines Java-Programms das Öffnen mehrerer Fenster – eines, um das Programm zu schreiben, ein anderes, um das Programm ablaufen zu lassen, und eventuell ein drittes, um dem ganzen Code, den Sie geschrieben haben, auf der Spur zu bleiben (siehe [Abbildung 2.4](#)).

Eine integrierte Entwicklungsumgebung (IDE) vereint diese Funktionen in einer gut durchdachten Anwendung (siehe [Abbildung 2.5](#)).



[Abbildung 2.4](#): Code-Entwicklung ohne integrierte Entwicklungsumgebung



[Abbildung 2.5](#): Code-Entwicklung mit der Entwicklungsumgebung Eclipse

Für Java gibt es einige integrierte Entwicklungsumgebungen. Zu den beliebtesten gehören Eclipse, IntelliJ IDEA und NetBeans. Einige der besseren Umgebungen kennen sogar »Drag and Drop«, was es Ihnen ermöglicht, Ihre grafischen Oberflächen unter Sichtkontrolle zu erstellen (siehe [Abbildung 2.6](#)).

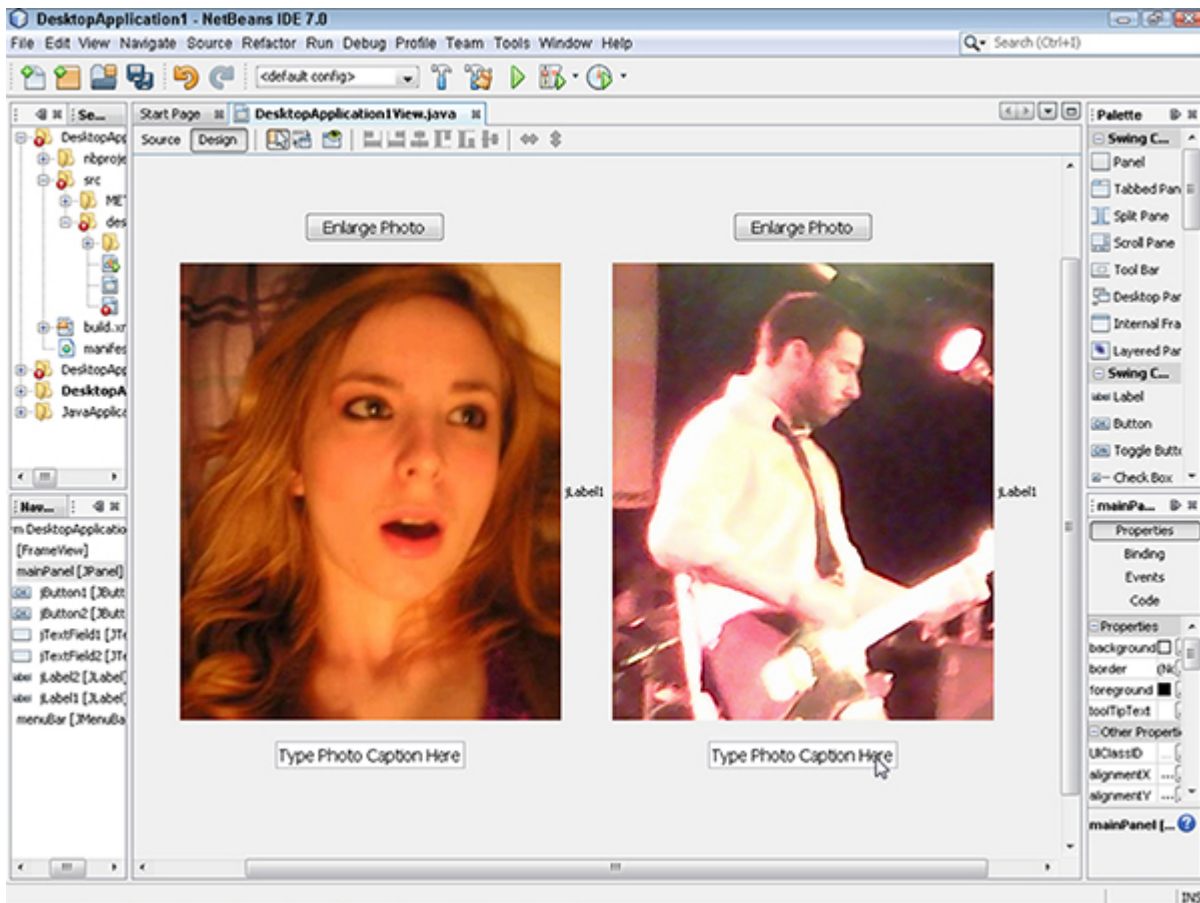


Abbildung 2.6: In NetBeans IDE den Swing GUI Builder für »Drag and Drop« verwenden

Um ein Programm auszuführen, müssen Sie normalerweise nur auf eine Schaltfläche in der Symbolleiste klicken oder in einem Menü RUN wählen (IDEs sind in der Regel englischsprachig). Um ein Programm zu kompilieren, müssen Sie eigentlich nichts tun. (Eventuell müssen Sie einen Befehl absetzen. Einige IDEs kompilieren Ihren Code automatisch, während Sie ihn eingeben.)

3

Die grundlegenden Teile verwenden

In diesem Kapitel

- ▶ Die Sprache Java sprechen: die API und die Besonderheiten der Sprache
 - ▶ Die Bestandteile eines einfachen Programms verstehen
 - ▶ Den Code dokumentieren
-

»Все мысли, которые имеют огромные последствия всегда просты.«

(Alle großen Ideen sind einfach.)

Leo Tolstoi

Dieses Zitat gilt für alle möglichen Dinge – Dinge wie das Leben, die Liebe und die Programmierung für Computer. Aus diesem Grund nähert sich dieses Kapitel diesem Thema auf mehreren Ebenen. Sie erhalten hier Ihre ersten Grundkenntnisse der Java-Programmierung, und während Sie die entsprechenden Einzelheiten entdecken, werden Sie überrascht sein, wie einfach sie gehalten sind.

Die Sprache Java sprechen

Wenn Sie versuchen, sich in Ihrem Kopf die gesamte deutsche Sprache bildhaft vorzustellen, was sehen Sie dann? Vielleicht Wörter, Wörter, Wörter. Wenn Sie sich eine Sprache unter dem Mikroskop anschauen, sehen Sie ein Wort nach dem anderen. Dieses Bild mit den vielen Wörtern geht in Ordnung, aber wenn Sie sich etwas zurücklehnen, fallen Ihnen vielleicht zwei andere Dinge auf:

- ✓ Die Grammatik der Sprache
- ✓ Tausende von Ausdrücken, Redewendungen, Sprichwörtern und historischen Bezeichnungen

Die erste Kategorie (die Grammatik) enthält Regeln wie: »Dieses Prädikat stimmt in Person und Numerus mit dem Substantiv überein.« Die zweite Kategorie (Ausdrücke, Redewendungen und so weiter) enthält Wissen wie: »Julius Caesar war ein bekannter römischer Kaiser. Gib deinem Sohn nur dann den Namen Julius Caesar, wenn du willst, dass er tagtäglich nach der Schule Prügel bezieht.«

Die Programmiersprache Java besitzt alle Bestandteile einer gesprochenen Sprache wie Deutsch. Und auch Java kennt Wörter, eine Grammatik, normalerweise verwendete Namen, Redensarten und andere Dinge dieser Art.

Die Grammatik und die gebräuchlichen Bezeichnungen

Die Leute von Sun Microsystems, die Java entwickelten, stellten sich ein zweiteiliges Java vor. So wie die deutsche Sprache eine eigene Grammatik und häufig verwendete Begriffe hat, kennt auch die Programmiersprache Java ihre Spezifikationen (ihre Grammatik) und ihr Application Programming Interface (ihre häufig genutzten Begriffe). Immer wenn ich Java-Programme schreibe, liegen zwei wichtige Dokumentationen auf meinem Schreibtisch – jeweils eine für die beiden Bereiche der Sprache:

- ✓ **Die Java-Sprachspezifikation:** Hierzu gehören Regeln wie:
 - »Setzen Sie immer eine öffnende Klammer hinter das Wort `for`« und »Verwenden Sie ein Sternchen, um zwei Zahlen miteinander zu multiplizieren«.
- ✓ **Das Application Programming Interface:** Javas Application Programming Interface (API; deutsch *Schnittstelle für die Anwendungsprogrammierung*) enthält Tausende von Werkzeugen, die Java hinzugefügt worden sind, nachdem die Grammatik der Sprache definiert worden ist. Diese Werkzeuge

lösen von ganz gewöhnlichen bis hin zu exotischen Aufgaben so gut wie alle. So gehört zum Beispiel zu den Werkzeugen eine Routine mit dem Namen *pow*, mit der Sie Zahlen potenzieren können. Ein auffälliges Werkzeug (mit dem Namen *JFrame*) zeigt auf dem Bildschirm des Computers ein Fenster an. Und wieder andere Werkzeuge achten darauf, ob ein Benutzer Schaltflächen anklickt, fragen Datenbanken ab und erledigen alle möglichen nützlichen Aufgaben.



Sie können die Java-Sprachspezifikation, die API-Dokumente und die gesamte Java-Dokumentation herunterladen (oder sich online anschauen), indem Sie

<http://docs.oracle.com/javase/specs> besuchen.

Aber aufgepasst! Bei dieser Webseite handelt es sich um ein bewegliches Ziel. Es kann passieren, dass diese Internetadresse nicht mehr stimmt, wenn Sie dieses Buch lesen. Deshalb ist es am sichersten, mit `Java.Sun.com` zu beginnen und dann nach Links Ausschau zu halten, die auf Dinge wie »Java SE« (der Kurzform für Java Standard Edition) und »reference« oder »documentation« verweisen. Die originalen Dokumentationen sind leider alle nur in Englisch vorhanden, aber unter

<http://de.wikibooks.org/wiki/Java> finden Sie mit ein wenig Suchen auch die entsprechende deutschsprachige Literatur.

Der erste Teil von Java, die Language Specification (die Sprachspezifikation), ist relativ klein. Das heißt nun aber nicht, dass Sie die Regeln der Sprachspezifikation im Handumdrehen erlernen können. Aber vielleicht tröstet es Sie, dass andere Programmiersprachen die doppelte, dreifache oder sogar zehnfache Anzahl an Regeln enthalten.

Der zweite Teil von Java – die API – kann wegen seines Umfangs abschreckend wirken. Die API enthält ungefähr 4.000 Werkzeuge und wächst mit jeder neuen Version von Java. Allerdings müssen Sie keine Funktion der API auswendig lernen. Nichts. *Nothing*. *Nada*. Sie können den Kram, den Sie benötigen, in der

Dokumentation nachlesen und das ignorieren, was Sie nicht benötigen. Was Sie oft verwenden, prägt sich von selbst in Ihr Gedächtnis ein. Was Sie nur selten anpacken, werden Sie (wie alle anderen Programmierer) schnell wieder vergessen.



Niemand kennt alle Funktionen der Java-API. Wenn Sie ein Java-Programmierer sind, der oft Programme schreibt, die neue Fenster öffnen, wissen Sie, wie Sie die API-Klasse `JFrame` zu verwenden haben. Wenn Sie nur selten Programme schreiben, die Fenster öffnen, können Sie die ersten paar Mal, wenn Sie Fenster benötigen, die Klasse `JFrame` in der API-Dokumentation nachlesen. Ich bin der Überzeugung, dass ein normaler Java-Programmierer, der darauf verzichtet, etwas in der API-Dokumentation nachzuschlagen, nur maximal zwei Prozent der Werkzeuge einsetzen kann, die es in der Java-API gibt.

In gewisser Weise enthält die Java-API nichts Besonderes. Jedes Java-Programm, das Sie schreiben – und sei es noch so klein und einfach –, enthält eine Klasse, die mit den Klassen der offiziellen Java-API auf einer Stufe steht. Die API ist nichts als eine Menge von Klassen und anderen Werkzeugen, die von ganz normalen Programmierern geschrieben wurden, die zufällig beim Java Community Process (JCP) und dem OpenJDK Project mitmachen. Die Werkzeuge in der API werden anders als die Werkzeuge, die Sie erstellen, mit jeder Java-Version weitergegeben. (Ich unterstelle dabei, dass Sie als Leser dieses Buches nicht beim Java Community Process oder dem OpenJDK Project mitmachen. Aber bei einem so tollen Buch wie *Java für Dummies* weiß man das nie.)



Wenn Sie an den Aktivitäten des JCP interessiert sind, besuchen Sie www.jcp.org. Wenn Sie mehr über das OpenJDK Project wissen wollen, besuchen Sie openjdk.java.net.



Die Leute vom JCP sorgen dafür, dass die Java-Programme in der offiziellen Java-API kein Geheimnis bleiben. Wenn Sie Interesse daran haben, können Sie sich diese Programme ansehen. Wenn Sie Java auf Ihrem Computer installieren, legt die Installationsroutine auf Ihrer Festplatte eine Datei mit dem Namen `src.zip` ab. Sie können diese Datei mit dem von Ihnen favorisierten Entpackungsprogramm öffnen. Und dann haben Sie den gesamten API-Code vor Augen.

Die Wörter in einem Java-Programm

Ein echter Java-Programmierer wird sagen, dass die Programmiersprache Java zwei verschiedene Arten von Wörtern kennt: Schlüsselwörter und Bezeichner. Das stimmt. Aber die nackte Wahrheit führt ohne eine weitere Erklärung gerne in die Irre. Deshalb möchte ich die Betrachtung etwas erweitern und von drei Arten von Wörtern ausgehen: Schlüsselwörter, Bezeichner, die normale Programmierer wie Sie und ich erstellen, und Bezeichner der API.

Die Unterschiede zwischen diesen drei Arten von Wörtern lassen sich mit den Unterschieden zwischen Wörtern in der deutschen Sprache vergleichen. Im Satz »Sam ist eine Person« entspricht das Wort *Person* einem Java-Schlüsselwort, unabhängig davon, wer das Wort *Person* verwendet, sagt es im Prinzip immer dasselbe aus. (Sicherlich könnten Sie sich jetzt einige bizarre Ausnahmen ausdenken, aber bitte tun Sie das nicht.)

Das Wort *Sam* entspricht einem Java-Bezeichner, weil Sam der Name einer bestimmten Person ist. Wörter wie *Sam*, *Kläusi* und *McGillimaroo* haben im Deutschen oder Englischen normalerweise keine festgelegte Bedeutung. Diese Wörter passen abhängig vom Umfeld auf verschiedene Personen und werden zu Namen, wenn Eltern sie für ihr neugeborenes Kind auswählen.

Betrachten Sie nun den Satz »Julius Caesar ist eine Person.« Wenn Sie diesen Satz aussprechen, reden Sie vielleicht über die

Person, die Rom bis zu den Iden des März regierte. Obwohl der Name *Julius Caesar* in der deutschen Sprache nicht fest verdrahtet ist, meint so gut wie jeder bei diesem Namen dieselbe Person. Wenn Deutsch eine Programmiersprache wäre, wäre der Name *Julius Caesar* ein API-Bezeichner.

Aus diesen Gründen unterteile ich die Wörter in einem Java-Programm in diese Kategorien:

✓ **Schlüsselwörter:** Ein *Schlüsselwort* ist ein Wort, das in der Programmiersprache Java seine eigene, ganz spezielle Bedeutung hat. Diese Bedeutung ändert sich nicht von Programm zu Programm. Beispiele für Java-Schlüsselwörter sind `if`, `else` und `do`.

Die Mitglieder des JCP-Komitees, die letztendlich bestimmen, was ein Java-Programm ausmacht, haben alle Java-Schlüsselwörter ausgewählt. Wenn Sie an die beiden Teile von Java denken, auf die ich weiter vorn in diesem Kapitel im Abschnitt *Die Grammatik und die gebräuchlichen Bezeichnungen* eingehe, gehören die Java-Schlüsselwörter zur Sprachspezifikation.

✓ **Bezeichner:** Ein *Bezeichner* ist ein Name für ein bestimmtes Element. Die Bedeutung eines Bezeichners kann sich von einem Programm zu einem anderen ändern, wobei sich die Bedeutungen einiger Bezeichner häufiger ändern können als die Bedeutung anderer Bezeichner.

- *Bezeichner, die Sie und ich erstellen:* Als Java-Programmierer (ja, sogar als Neuling unter den Java-Programmierern) legen Sie neue Namen für Klassen und andere Elemente fest, die Sie in Ihren Programmen beschreiben. So können nicht nur Sie etwas als `Tor` bezeichnen, sondern auch der Kollege, der zwei Schreibtische weiter ebenfalls Code schreibt, gibt etwas den Namen `Tor`. Das geht in Ordnung, weil es in Java keine vorgefertigte Bedeutung für das Wort `Tor` gibt. In Ihrem Programm steht `Tor` für den Zugang zu einem Grundstück, während Ihr sportbegeisterter Kollege `Tor` für den »Kasten« auf dem Fußballplatz verwendet. Es kommt zu keinem Konflikt, weil

Sie und Ihr Kollege zwei verschiedene Java-Programme schreiben.

- *Bezeichner der API:* Die JCP-Mitglieder haben sich für viele Dinge Namen einfallen lassen und ungefähr 40.000 davon in der Java-API untergebracht. Die Java-API wird immer zusammen mit jeder Java-Version ausgeliefert, weshalb diese Namen für jeden zur Verfügung stehen, der ein Java-Programm schreibt. Beispiele für solche Namen sind `String`, `Integer`, `JWindow`, `JButton`, `JPasswordField` und `File`.

Genau genommen sind die Bedeutungen der Bezeichner in der Java-API nicht in Stein gemeißelt. Aber selbst wenn Sie `JButton` eine eigene Bedeutung geben können, ist dies keine besonders gute Idee. Wenn Sie so vorgehen, verwirren Sie nur andere Programmierer, die sich an die Standard-API-Bedeutungen dieser bekannten Bezeichner gewöhnt haben. Noch unangenehmer aber ist, dass Sie die gesamte Leistung verlieren, die sich im API-Code für den Bezeichner wie `JButton` versteckt, wenn Sie ihm eine neue Bedeutung zuweisen. Die Programmierer von Sun Microsystems, der Java Community Process und das OpenJDK Project haben die gesamte Arbeit erledigt und Java-Code geschrieben, damit Sie problemlos mit Schaltflächen (englisch *Buttons*) umgehen können. Wenn Sie `JButton` eine neue Bedeutung zuweisen, verzichten Sie auf alle Vorteile, die das Arbeiten mit der API bietet.

Sich zum ersten Mal mit Java-Code beschäftigen

Wenn Sie sich zum ersten Mal das Java-Programm eines fremden Programmierers anschauen, fühlen Sie sich vielleicht etwas unwohl. Die Erkenntnis, dass Sie den gesamten Code (oder zumindest vieles davon) nicht verstehen, macht Sie möglicherweise nervös. Ich habe Hunderte (wenn nicht sogar Tausende) von Java-Programmen geschrieben, aber ich bin immer noch unsicher, wenn ich den Code eines anderen lese.

Tatsächlich ist es so, dass die Analyse eines Java-Programms einer Volltextsuche ähnelt. Zuerst blicken Sie das Programm ehrfurchtsvoll an. Dann führen Sie das Programm aus, um zu sehen, was es macht. Dann starren Sie das Programm eine Weile an oder lesen sich das durch, was irgendjemand als Erklärung zum Programm und seinen Komponenten geschrieben hat. Dann werfen Sie einen noch intensiveren Blick auf das Programm und führen es erneut aus. Und vielleicht kommen Sie dann irgendwann auch mit dem Programm klar. (Glauben Sie nicht den angeblichen Experten, die Ihnen weismachen wollen, dass sie diese Schritte nicht durchlaufen. Selbst die erfahrensten Programmierer gehen ein neues Projekt langsam und sorgfältig an.)

Mit [Listing 3.1](#) erhalten Sie ein hervorragendes Beispiel eines Java-Codes. (Und ich erwarte von Ihnen, dass Sie sich wie alle Programmierneulinge verhalten und dieses Beispiel ehrfurchtsvoll behandeln.) Ich habe in diesem Code einige wichtige Ideen versteckt, die ich im nächsten Abschnitt im Einzelnen beschreibe – zum Beispiel der Gebrauch von Klassen, Methoden und Java-Befehlen.

```
public class Displayer {  
    public static void main(String args[]) {  
        System.out.println("Sie werden Java lieben!");  
    }  
}
```

[Listing 3.1](#): Ein einfaches Java-Programm

Wenn Sie das Programm in [Listing 3.1](#) ausführen, gibt der Computer Sie werden Java lieben! aus. ([Abbildung 3.1](#) zeigt, was das Programm `Displayer` wiedergibt, wenn Sie die Eclipse-IDE verwenden.) Zugegeben, es bereitet viel Arbeit, ein Java-Programm zu schreiben und auszuführen, nur um Sie werden Java lieben! auf dem Bildschirm eines Computers auszugeben, aber irgendwo müssen Sie bekanntlich anfangen.



Wenn Sie nicht wissen, wie Sie den Code in [Listing 3.1](#) ausführen können, besuchen Sie entweder www.java-programmieren.com/eclipse-erste-schritte.php oder die amerikanische Webseite des Autors zu diesem Buch (allmycode.com/JavaForDummies).



Abbildung 3.1: Ich verwende Eclipse, um das Programm in [Listing 3.1](#) auszuführen.

Im nächsten Abschnitt machen Sie mehr, als nur die Ausgabe des Programms zu bewundern. Wenn Sie ihn gelesen haben, sind Sie in der Lage zu verstehen, wieso das Programm in [Listing 3.1](#) funktioniert.

Ein einfaches Java-Programm verstehen

Dieser Abschnitt präsentiert, erklärt, analysiert, seziert und entzaubert das Java-Programm, das [Listing 3.1](#) zeigt.

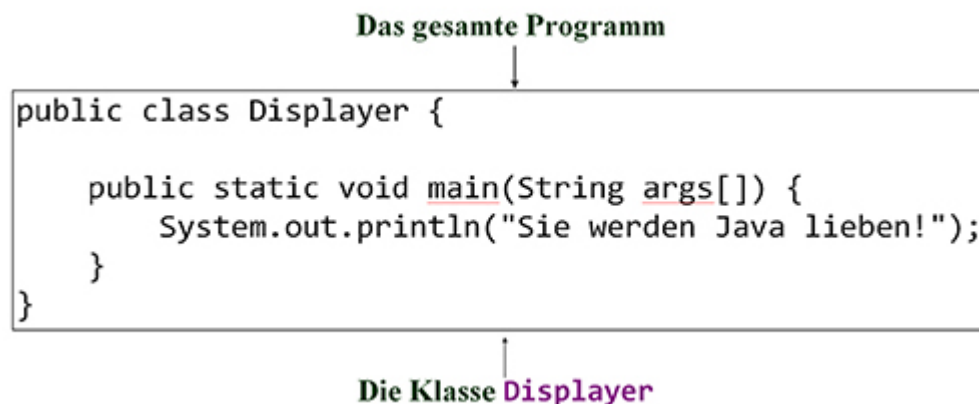
Die Java-Klasse

Da Java eine objektorientierte Programmiersprache ist, ist es Ihre vorrangige Aufgabe, Klassen und Objekte zu beschreiben. (Wenn Ihnen das nicht ganz klar ist, sollten Sie den Abschnitt über objektorientierte Programmierung in [Kapitel 1](#) lesen.)

Java weist einen viel höheren Grad an Objektorientierung auf als die meisten anderen der sogenannten objektorientierten

Sprachen. Sie können in Java erst dann etwas machen, wenn Sie eine Klasse angelegt haben.

In Java bildet das gesamte Programm eine Klasse. Da ich das Programm geschrieben habe, kann ich der neuen Klasse auch einen Namen meiner Wahl geben. Ich habe mich für den Namen `Display` (deutsch *Anzeiger*) entschieden, weil das Programm auf dem Bildschirm des Computers eine Textzeile anzeigt. Aus diesem Grund enthält die erste Zeile in [Listing 3.1](#) auch die Wörter `class Display` (siehe [Abbildung 3.2](#)).



[Abbildung 3.2](#): Ein Java-Programm ist eine Klasse.

Die ersten beiden Wörter in [Listing 3.1](#), `public` und `class`, sind Java-Schlüsselwörter (siehe hierzu auch weiter vorn in diesem Kapitel den Abschnitt *Die Wörter in einem Java-Programm*). Unabhängig davon, wer ein Java-Programm schreibt, werden die Wörter `public` und `class` immer auf dieselbe Weise verwendet. Dagegen ist `Display` ein Bezeichner. Mir ist das Wort `Display` in den Sinn gekommen, als ich dieses Kapitel geschrieben habe. `Display` ist der Name einer Klasse – der Klasse, die ich erstelle, indem ich dieses Programm schreibe. (Dieses Buch ist voll mit Informationen über Klassen, aber eine ausführliche Beschreibung dessen, was eine Klasse ist und wie sie verwendet wird, finden Sie in [Kapitel 7](#). Das Wort `public` bedeutet (einfach ausgedrückt), dass jedes andere Java-

Programm die so deklarierte Funktion nutzen kann. Weitere Informationen zu diesem Thema gibt es in den [Kapiteln 7](#) und [14](#).)



DIE PROGRAMMIERSPRACHE JAVA BEACHTET GROSS- UND KLEINSCHREIBUNG. WENN SIE EIN WORT VON kleinbuchstaben IN GROSSBUCHSTABEN ODER VON GROSSBUCHSTABEN IN kleinbuchstaben ÄNDERN; ÄNDERN SIE GLEICHZEITIG DIE BEDEUTUNG DIESES WORTES. SO KÖNNEN SIE ZUM BEISPIEL IN Listing 3.1 class NICHT DURCH Class ERSETZEN. WÜRDEN SIE DIES tun, würde DAS GESAMTE PROGRAMM AUFHÖREN ZU FUNKTIONIEREN. Ähnliches gilt bis zu einem gewissen Grad auch für den Namen einer Datei, die eine bestimmte Klasse enthält. So ist zum Beispiel der Name der Klasse in [Listing 3.1](#) `DisplayeR`. Er beginnt mit einem großen D. Es wäre deshalb richtig gut, den Code in [Listing 3.1](#) in einer Datei zu speichern, die `DisplayeR.java` heißt und deren Name ebenfalls mit einem großen D beginnt.

Die Java-Methode

Sie arbeiten als Mechaniker in einer Werkstatt für exklusive Autos. Ihr Chef hat es immer eilig und hängt beim Sprechen Wörter zusammen. Er ruft Ihnen zu: »WechsleDieLichtmaschine bei demAltenFord.« Sie stellen sich im Geiste vor, welche Aufgaben Sie zu erledigen haben: »Auto in die Halle fahren, Motorhaube öffnen, einen Schraubenschlüssel nehmen, den Keilriemen lösen und so weiter.« Hier laufen drei Dinge ab:

- ✓ **Es gibt für das, was Sie vorhaben, einen Namen.** Der Name ist `wechsleDieLichtmaschine`.
- ✓ **Im Kopf haben Sie eine Liste mit Aufgaben, die mit dem Namen `wechsleDieLichtmaschine` verbunden sind.** Zu dieser Liste gehören: »Fahre das Auto in die Halle, öffne die Motorhaube, nimm einen Schraubenschlüssel, löse den Keilriemen« und so weiter.

✓ **Sie haben einen Chef, der diese Arbeit anordnet.** Ihr Chef bekommt Sie dadurch ans Arbeiten, dass er sagt:
»WechsleDieLichtmaschine.« Sie arbeiten, weil Ihr Chef den Namen der Sache ausspricht, die Sie erledigen sollen.

In dieser Situation ist die Verwendung des Wortes *Methode* nicht zu weit hergeholt. Sie kennen eine Methode, um etwas mit einer Lichtmaschine zu tun. Ihr Chef bringt diese Methode ins Spiel, und Sie reagieren darauf, indem Sie die Dinge erledigen, die Sie mit der Methode verbinden.

Wenn Sie das alles glauben (was ich hoffe), sind Sie so weit, dass Sie etwas über Java-Methoden erfahren können. In Java ist eine *Methode* eine Liste mit Dingen, die erledigt werden müssen. Jede Methode hat einen Namen und Sie weisen den Computer an, die Dinge aus der Liste zu erledigen, indem er den Namen verwendet, den die Methode in Ihrem Programm hat. Ich habe noch kein Programm geschrieben, das einen Roboter dazu bringt, eine Lichtmaschine zu wechseln. Aber sollte ich das einmal tun, könnte dieses Programm die Methode `wechsleDieLichtmaschine` enthalten. Und die Liste mit den Anweisungen in dieser Methode würde so ähnlich aussehen wie der Text in [Listing 3.2](#).



Studieren Sie die [Listings 3.2](#) und [3.3](#) nicht zu genau. Der gesamte Code in diesen beiden Listings ist erfunden! Er sieht wie Java-Code aus, ist es aber nicht. Außerdem ist es wichtig, dass Sie wissen, dass der Code in den [Listings 3.2](#) und [3.3](#) keine Java-Regeln zeigen soll. Aufgabe dieser beiden Listings, die Sie nicht auf eine Goldwaage legen dürfen, ist es, den Zusammenhang zwischen Methodendeklaration und Methodenaufruf zu zeigen.

```
void wechselDieLichtmaschine() {  
    driveInto(auto, halle);  
    lift(motorhaube);  
    get(schraubenschlüssel);  
    loosen(keilRiemen);  
    ...  
}
```

[Listing 3.2](#): Eine Methodendeklaration

Irgendwo in meinem Java-Code (irgendwo außerhalb von [Listings 3.2](#)) benötige ich eine Anweisung, um meine Methode `wechsleDieLichtmaschine` aufzurufen. Diese Anweisung könnte so aussehen wie in [Listings 3.3](#).

```
wechsleDieLichtmaschine(alterFord);
```

[Listing 3.3](#): Ein Methodenaufruf

Nachdem Sie nun grundlegend wissen, was eine Methode ist und wie sie arbeitet, können Sie ein wenig tiefer in die entsprechende Terminologie eintauchen:

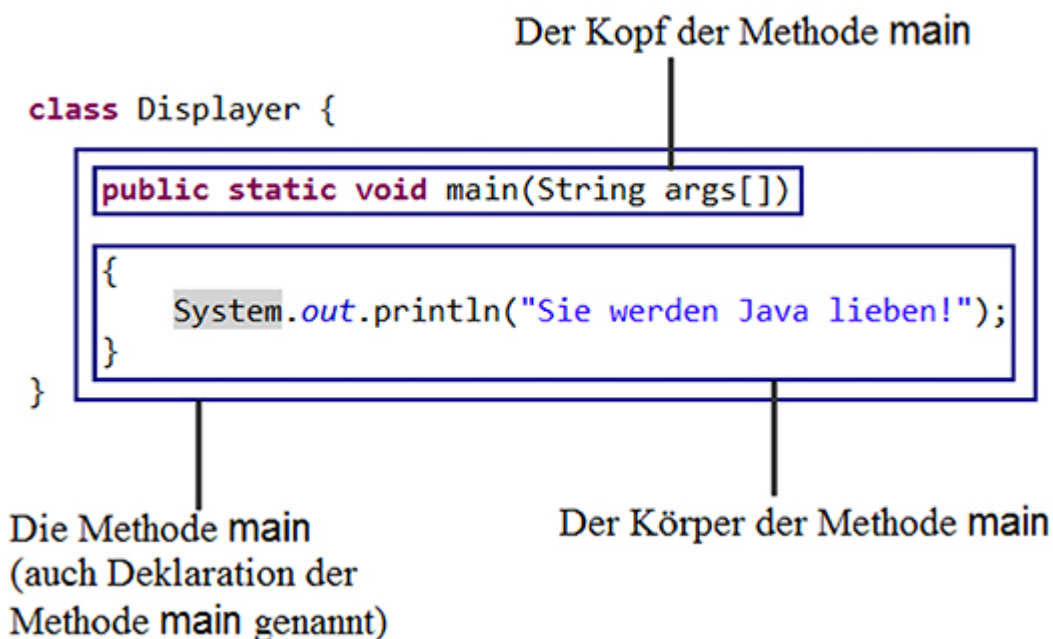
- ✓ Wenn ich es mir bequem mache, bezeichne ich den Code in [Listing 3.2](#) als *Methode*, obwohl es sich dabei eigentlich um eine *Methodendeklaration* handelt.
- ✓ Die Methodendeklaration in [Listing 3.2](#) besteht aus zwei Teilen. Die erste Zeile (der Teil von `wechsleDieLichtmaschine` bis (aber nicht einschließlich) zur öffnenden geschweiften Klammer wird als *Methodenkopf* bezeichnet, während der Rest (der Teil, der von den geschweiften Klammern eingeschlossen wird) *Methodenkörper* genannt wird.
- ✓ Der Begriff *Methodendeklaration* unterscheidet zwischen der Liste mit Anweisungen in [Listing 3.2](#) und den Anweisungen in [Listing 3.3](#), die *Methodenaufruf* genannt werden.



Die *Deklaration einer Methode* sagt dem Computer, was geschehen soll, wenn Sie die Methode aufrufen. Ein *Methodenaufruf* (ein eigenständiges Stückchen Code) weist den Computer an, die Methode tatsächlich zu aktivieren. Eine Methodendeklaration und ein Methodenaufruf befinden sich normalerweise in unterschiedlichen Teilen des Java-Programms.

Die Methode »main« eines Programms

[Abbildung 3.3](#) zeigt eine Kopie von [Listing 3.1](#). Der größte Teil des Codes enthält die Deklaration einer Methode, die den Namen `main` trägt. (Suchen Sie im Methodenkopf nach dem Wort *main*.) Ignorieren Sie im Moment noch die übrigen Wörter im Methodenkopf – `public`, `static`, `void`, `String` und `args`. Ich erkläre diese Begriffe im nächsten Abschnitt.



[Abbildung 3.3](#): Die Methode *main*

Die Methode `main` ist – wie jede Java-Methode – ein Rezept.

Wie Kekse gebacken werden

Heizen Sie den Ofen vor.

Rollen Sie den Teig aus.

Backen Sie den ausgerollten Teig.

Oder

Wie den primären Anweisungen für einen

`Displayer` zu folgen ist:

Gib auf dem Bildschirm "Sie werden Java lieben!"
aus.

Das Wort *main* spielt in Java eine besondere Rolle. (Dieses Wort bedeutet im Deutschen *primär*, *wesentlich*, *hauptsächlich* oder *zentral*.) Sie dürfen auf keinen Fall Code schreiben, der ausdrücklich eine Methode mit dem Namen `main` aufruft. Bei dem Wort *main* handelt es sich um den Namen der Methode, die automatisch aktiviert wird, wenn das Programm mit seiner Ausführung beginnt.

Schauen Sie sich noch einmal [Abbildung 3.1](#) an. Wenn das Programm `Displayer` läuft, findet der Computer die Methode `main` des Programms automatisch und führt jede Anweisung aus, die sich im Körper dieser Methode befindet. Im Programm `Displayer` gibt es im Körper der Methode `main` nur eine Anweisung. Diese sagt dem Computer, dass er `Sie werden Java lieben!` auf dem Bildschirm ausgeben soll. Was, wie [Abbildung 3.1](#) zeigt, auch geschieht.



Anweisungen einer Methode werden erst dann ausgeführt, wenn die Methode aufgerufen wird. Wenn Sie einer Methode aber den Namen *main* geben, wird diese Methode automatisch aufgerufen.



So gut wie jede Programmiersprache kennt so etwas wie die Java-Methoden. Wenn Sie bereits mit anderen Sprachen gearbeitet haben, erinnern Sie sich vielleicht an Dinge wie Unterprogramme, Prozeduren, Funktionen, Unterrouinen, Unterprozeduren oder `PERFORM`-Befehle. In jedem Fall handelt es sich um eine Gruppe von Anweisungen, die einen eigenen Namen bekommen haben.

Wie Sie dem Computer letztendlich mitteilen, dass er etwas tun soll

Tief im Herzen von [Listing 3.1](#) vergraben befindet sich die einzige Zeile, die letztendlich eine direkte Anweisung an den Computer

enthält. Diese Zeile, die in [Abbildung 3.4](#) optisch hervorgehoben worden ist, weist den Computer an, Sie werden Java lieben! anzuzeigen. Eine solche Zeile wird als *Befehl* bezeichnet. In Java ist ein Befehl eine direkte *Anweisung*, die den Computer auffordert, etwas zu tun (zum Beispiel diesen Text anzuzeigen, an einer Speicherposition 7 einzufügen, ein Fenster erscheinen zu lassen).

```
class Displayer {  
  
    public static void main(String args[])  
    {  
        System.out.println("Sie werden Java lieben!");  
    }  
}
```

Ein Befehl (ein Aufruf der Methode System.out.println)

[Abbildung 3.4](#): Ein Java-Befehl



In `System.out.println` ist der vorletzte Buchstabe ein kleines *l* (das »el« ausgesprochen wird) und keine Ziffer 1 (eins).

Natürlich kennt Java verschiedene Arten von Befehlen. Ein Methodenaufruf, den ich weiter vorn in diesem Kapitel im Abschnitt *Die Java-Methode* vorstelle, ist einer der vielen möglichen Java-Befehle. [Listing 3.3](#) zeigt Ihnen, wie ein Methodenaufruf aussieht, und [Abbildung 3.4](#) enthält einen Methodenaufruf, der so aussieht:

```
System.out.println("Sie werden Java lieben!");
```

Wenn der Computer diesen Befehl ausführt, ruft er eine Methode auf, die `System.out.println` heißt. (Ja, in Java kann ein

Name Punkte enthalten. Und diese Punkte haben eine besondere Bedeutung, auf die in [Kapitel 9](#) genauer eingegangen wird.)

[Abbildung 3.5](#) veranschaulicht den Aufruf von `System.out.println`. Tatsächlich sind zwei Methoden aktiv an der Ausführung des Programms `Displayer` beteiligt:

✓ **Es gibt eine Deklaration der Methode `main`.** Ich habe die Methode `main` selbst geschrieben. Sie wird automatisch aufgerufen, wenn ich das Programm `Displayer` ausführe.

✓ **Es gibt einen Aufruf der Methode `System.out.println`.** Der Methodenaufruf von `System.out.println` ist der einzige Befehl in der Aufgabenliste der Methode `main`.

Die Deklaration der Methode `System.out.println` ist irgendwo in der offiziellen Java-API begraben. Wenn Sie Ihre Kenntnisse über die Java-API auffrischen wollen, schauen Sie sich noch einmal weiter vorn in diesem Kapitel die Abschnitte *Die Grammatik und die gebräuchlichen Bezeichnungen* und *Die Wörter in einem Java-Programm* an.

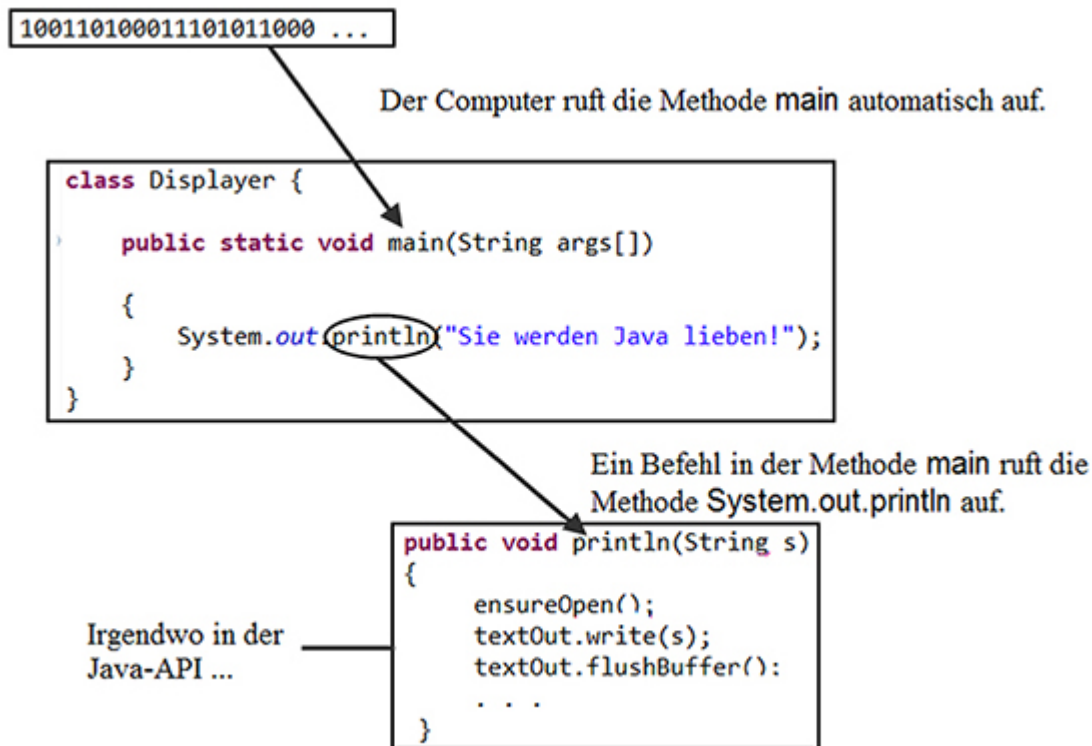


Abbildung 3.5: Die Methode `System.out.println` wird aufgerufen.



Wenn ich Dinge wie »Die Deklaration der Methode `System.out.println` ist irgendwo in der offiziellen Java-API begraben« schreibe, werde ich der API nicht gerecht. Klar, Sie können natürlich den gesamten praktischen Java-Code ignorieren, den es in der API gibt. Sie müssen sich nur daran erinnern, dass `System.out.println` irgendwo im Code definiert wird. Aber ich bin nicht fair, wenn ich so tue, als ob der API-Code etwas Magisches sei. Bei der API handelt es sich um nichts anderes als um eine Menge von Java-Code. Die Befehle in der API, die dem Computer sagen, was es bedeutet, einen Aufruf von `System.out.println` auszuführen, haben viel Ähnlichkeit mit dem Code in [Listing 3.1](#)

In Java wird jeder Befehl (wie der hervorgehobene in [Abbildung 3.4](#)) mit einem Semikolon abgeschlossen. Die anderen Zeilen in

[Abbildung 3.4](#) enden nicht mit einem Semikolon, weil sie keine Befehle sind. So gibt zum Beispiel der Methodenkopf (die Zeile, die das Wort *main* enthält) dem Computer keine Anweisung, etwas zu tun. Der Methodenkopf verkündet eigentlich nur: »Falls du irgendwann einmal auf die Idee kommen solltest, `main` auszuführen, sagen dir die nächsten Codezeilen, wie das dann erledigt werden muss.«



Jeder vollständige Java-Befehl wird mit einem Semikolon beendet.

Geschweifte Klammern

Vor mehr oder weniger langer Zeit hat sicherlich auch Ihr Lehrer versucht, Sie davon zu überzeugen, wie sinnvoll es ist, einen Aufsatz mit einer Gliederung zu beginnen. Mit einer Gliederung können Sie Gedanken und Ideen ordnen, den Wald und nicht nur die Bäume sehen und ganz allgemein zeigen, dass Sie ein Mitglied des *Klubs der ordentlichen Menschen* sind. Ein Java-Programm ist wie eine Gliederung. Das Programm in [Listing 3.1](#) beginnt mit einer großen Überschrift, die den Beginn der Klasse `Display` anzeigt. Danach verkündet eine untergeordnete Überschrift, dass nun eine Methode mit dem Namen `main` folgt.

Wenn nun aber ein Java-Programm so etwas Ähnliches wie eine Gliederung ist, warum sieht es dann nicht auch wie eine Gliederung aus? Was nimmt den Platz von römischen Ziffern, Großbuchstaben und Ähnlichem ein? Die Antwort ist zweigeteilt:

- ✓ Code-Einheiten werden in einem Java-Programm von geschweiften Klammern eingeschlossen.
- ✓ Sie (der Programmierer) können (und sollten) Zeilen so einrücken, dass andere Programmierer die Gliederungsstruktur Ihres Codes auf einen Blick erkennen können.

Bei einer Gliederung ist alles dem Element mit der römischen Ziffer I untergeordnet. In einem Java-Programm ist alles der obersten Zeile untergeordnet – der Zeile mit `class`. Um anzuzeigen, dass der Rest des Codes dieser Zeile untergeordnet

ist, verwenden Sie geschweifte Klammern. Alles, was es im Programm sonst noch an Code gibt, gehört in diese geschweiften Klammern (siehe [Listing 3.4](#)).

```
public class Displayer {  
    public static void main(String args[]) {  
        System.out.println("Sie werden Java lieben!");  
    }  
}
```

[Listing 3.4](#): Geschweifte Klammern einer Java-Klasse

Bei einer Gliederung gibt es Einträge, die dem Großbuchstaben A untergeordnet sind (zum Beispiel *I. A.*). In einem Java-Programm werden einige Zeilen dem Methodenkopf untergeordnet. Um anzuzeigen, dass etwas dem Methodenkopf untergeordnet ist, verwenden Sie wieder geschweifte Klammern (siehe Listing 3.5).

```
public class Displayer {  
    public static void main(String args[]) {  
        System.out.println("Sie werden Java lieben!");  
    }  
}
```

[Listing 3.5](#): Geschweifte Klammern einer Java-Methode

Bei einer Gliederung gibt es Einträge, die am Ende der Nahrungskette stehen. In der Klasse `Displayer` beginnt die entsprechende Zeile mit `System.out.println`. Diese Zeile wird somit von allen geschweiften Klammern eingeschlossen und tiefer als alle anderen Zeilen eingerückt.

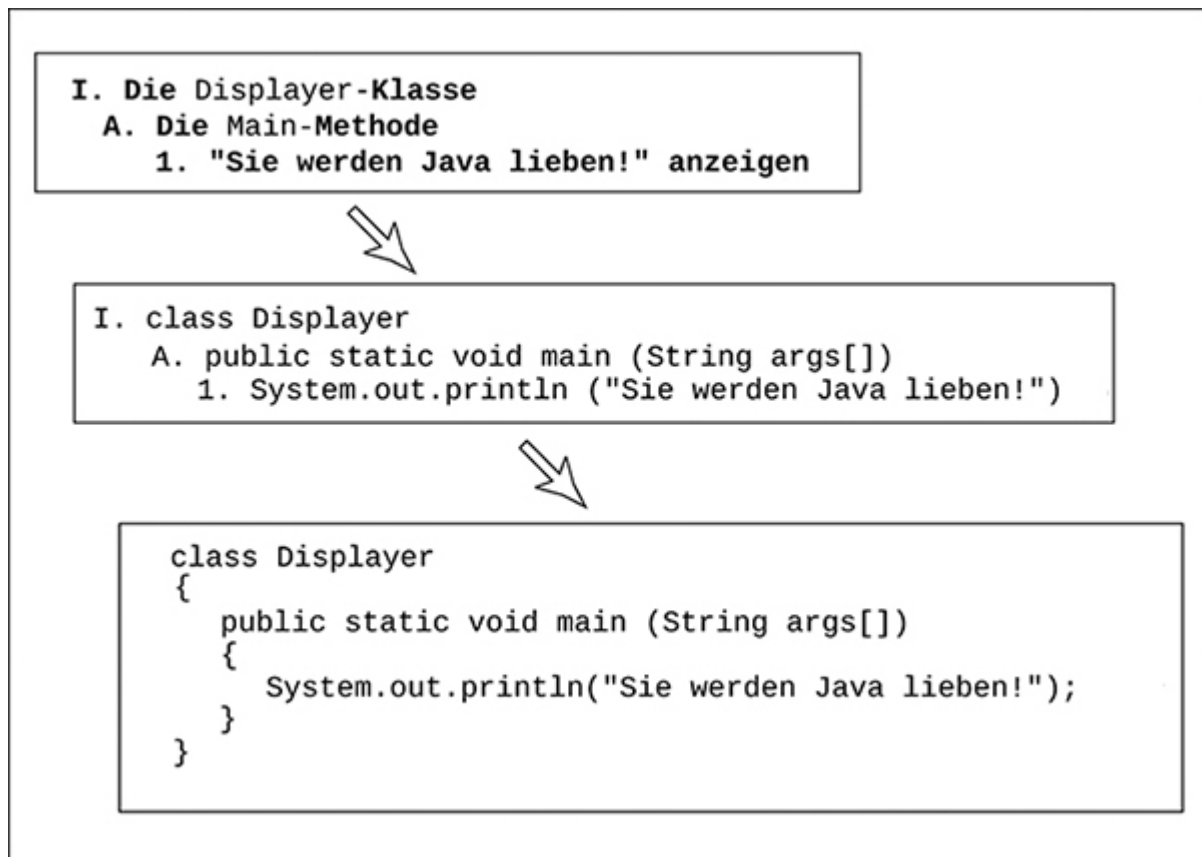


Vergessen Sie niemals, dass ein Java-Programm zunächst nichts als eine Gliederung ist.

Wenn Sie geschweifte Klammern an der falschen Stelle setzen oder vergessen, wohin sie gehören, funktioniert Ihr Programm höchstwahrscheinlich nicht. Sollte es aber laufen, ist es mit ziemlicher Sicherheit fehlerhaft.

Wenn Sie darauf verzichten, Zeilen einzurücken, um die Struktur des Programms zu verdeutlichen, funktioniert Ihr Programm zwar problemlos, aber weder Sie noch ein anderer Programmierer wird später in der Lage sein herauszubekommen, was Sie sich beim Schreiben des Codes gedacht haben.

Wenn Sie bildhaft denken, können Sie sich auch die Struktur eines Java-Programms bildhaft vorstellen. Einer meiner Freunde ist in der Lage, sich eine nur nummerierte Gliederung bildhaft vorzustellen und in ein Java-Programm zu übersetzen (siehe [Abbildung 3.6](#)). Jemand anders, der namenlos bleiben möchte, verwendet noch bizarrere Bilder (siehe [Abbildung 3.7](#)).



[Abbildung 3.6](#): Aus einer Gliederung wird ein Java-Programm.

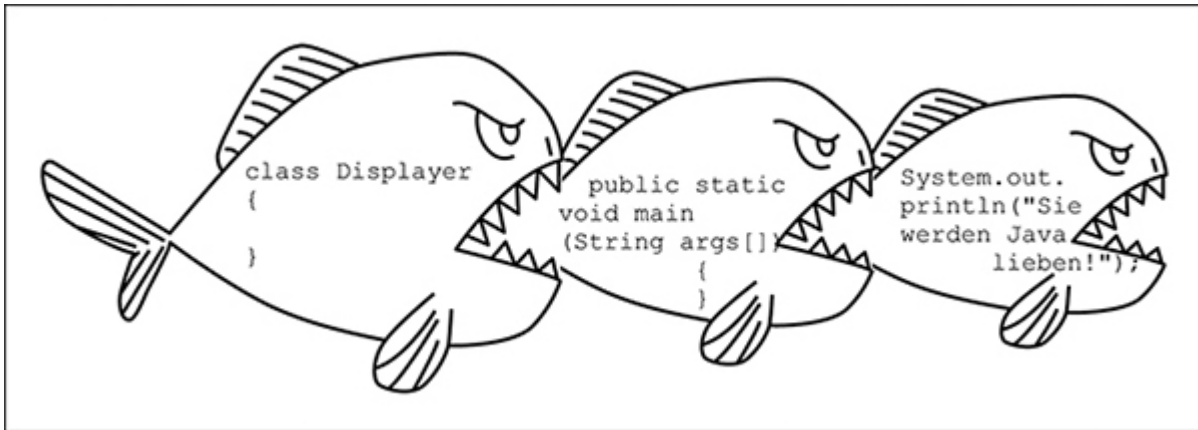


Abbildung 3.7: Eine Klasse ist größer als eine Methode, und eine Methode ist größer als ein Befehl.

Kommentare, Kommentare

Die Gruppe hat sich am Lagerfeuer versammelt, um die Geschichte über die Programmiererin zu hören, deren Bequemlichkeit sie in große Schwierigkeiten gebracht hat. Damit ihre Anonymität gewahrt bleibt, nennen wir sie einfach Jane Pro. Jane arbeitete viele Monate daran, den Heiligen Gral des Computerwesens zu entwickeln – ein Programm, das selbstständig denken kann. Wenn es fertiggestellt ist, kann es selbstständig arbeiten und ohne menschliche Anleitung neue Dinge erlernen. Sie schuftete Tag für Tag und Nacht für Nacht, um dem Programm den letzten Funken mitzugeben, der das kreative, unabhängige Denken zum Leuchten bringt.

Eines Tages, als sie mit dem Projekt fast fertig war, erhielt sie von ihrer Krankenkasse einen beunruhigenden Brief. Das Schreiben hatte nichts mit einer ernsthaften Erkrankung zu tun, es ging nur um einen routinemäßigen Besuch beim Arzt. Auf dem Formular für die Rückerstattung des vorauslagten Arzthonorars gab es ein Feld für das Geburtsdatum (als wenn sich das seit dem letzten Mal geändert hätte). Sie war beim Ausfüllen des Formulars mit den Gedanken ganz woanders gewesen und hatte als Geburtsjahr 2014 eingetragen, was dazu führte, dass sich die Versicherung weigerte zu zahlen.

Jane wählte die Telefonnummer der Versicherungsgesellschaft, und schon nach 20 Minuten hatte sie es geschafft, mit einer lebenden Person zu sprechen. »Das tut mir leid«, sagte diese Person, »aber Sie müssen eine andere Telefonnummer wählen.« Sie können sich vorstellen, wie es weiterging. »Das tut mir leid. Sie haben leider die falsche Telefonnummer erhalten.« Und: »Das tut mir leid. Sie müssen die Nummer anrufen, die Sie zuerst gewählt haben.«

Fünf Monate später schmerzte Janes Ohr, aber nach 800 Stunden am Telefon hatte sie es wenigstens geschafft, die unverbindliche Zusage zu erhalten, dass sich die Versicherungsgesellschaft den Vorgang eventuell noch einmal vornimmt. Beschwingt wie sie gerade war, begab sich Jane wieder an ihr Programmierprojekt. Konnte sie sich noch daran erinnern, was diese Vielzahl von Codezeilen machen sollten?

Nein, das konnte sie nicht. Sie starrte auf ihr eigenes Werk, und wie bei einem Traum, den man am nächsten Morgen verloren hat, sagte ihr der Code überhaupt nichts mehr. Sie hatte eine Million Codezeilen geschrieben, von denen keine von einem informativen Kommentar begleitet wurde. Sie hatte keine Hinweise hinterlassen, die ihr helfen konnten, ihre Gedankengänge zu verstehen. Frustriert gab sie das ganze Projekt auf.

Dem Code Kommentare hinzufügen

[Listing 3.6](#) ist eine erweiterte Version des Beispielprogramms dieses Kapitels. Das Listing enthält zusätzlich zu den Schlüsselwörtern, Bezeichnern und der Interpunktion Text, der für Menschen hinzugefügt worden ist.

```
/*
 * Listing 3.6 - "Java für Dummies"
 *
 * Copyright 2014 Wiley-VCH Verlag GmbH & Co. KG.
 * Alle Rechte vorbehalten.
 */

/**
 * Die Klasse Displayer zeigt Text auf dem
```

```

    * Bildschirm des Computers an.
    *
    * @author Barry Burd
    * @version 1.0 10/24/0913
    * @see java.lang.System
    */
public class Displayer {
    /**
     * Die Ausführung des Codes beginnt dort,
     * wo sich die Methode main befindet.
     *
     * @param args (siehe Kapitel 11.)
     */
    public static void main(String args[]) {
        System.out.println("Ich liebe Java!"); //Ich?
    }
}

```

[Listing 3.6](#): Die drei Kommentararten

Ein Kommentar ist ein besonderer Textbereich in einem Programm. Seine Aufgabe ist es, Menschen dabei zu helfen, das Programm zu verstehen. Ein Kommentar ist Teil einer guten Programmdokumentation.

Die Programmiersprache Java kennt drei Arten von Kommentaren:

✓ **Herkömmliche Kommentare:** In [Listing 3.6](#) bilden die ersten fünf Zeilen einen herkömmlichen Kommentar. Dieser beginnt mit `/*` und endet mit `*/`. Alles, was sich zwischen `/*` und `*/` befindet, ist nur für menschliche Augen bestimmt. Angaben wie "Java für Dummies" und Wiley-VCH Verlag GmbH & Co. KG werden vom Compiler nicht übersetzt. (Informationen über Compiler finden Sie in [Kapitel 2](#).)

Die Zeilen zwei, drei, vier und fünf haben in [Listing 3.6](#) zusätzliche Sternchen (*) erhalten. Ich spreche von zusätzlich, weil diese Sternchen nicht notwendig sind, wenn Sie einen Kommentar erstellen. Sie sorgen nur dafür, dass der Kommentar nett aussieht. Ich habe diese Sternchen in [Listing](#)

[3.6](#) hinzugefügt, weil die meisten Java-Programmierer sie aus Gründen hinzufügen, die ich nicht kenne,

✓ **Zeilenendekommentare:** In [Listing 3.6](#) ist der Text `//Ich? Sie?` ein Kommentar am Ende einer Zeile. Diese Art von Kommentar beginnt mit zwei Schrägstrichen und reicht bis zum Ende einer Zeile. Auch hier gilt, dass der Compiler den Text im Kommentar nicht übersetzt.

✓ **Javadoc-Kommentare:** Ein *Javadoc*-Kommentar beginnt mit einem Schrägstrich und zwei Sternchen (`/**`). In [Listing 3.6](#) gibt es zwei Javadoc-Kommentare – einen mit dem Text `Die Klasse Displayer ...` und einen zweiten mit dem Text `Die Ausführung des Codes ....`

Ein Javadoc-Kommentar ist eine Sonderform des herkömmlichen Kommentars. Er ist für jemanden gedacht, der sich noch nie mit Java-Code beschäftigt hat. Das ist aber ein Widerspruch in sich. Wie können Sie den Javadoc-Kommentar in [Listing 3.6](#) lesen, wenn Sie sich [Listing 3.6](#) nicht ansehen?

Nun, es gibt ein Programm mit dem Namen *javadoc* (klar!), das alle Javadoc-Kommentare in [Listing 3.6](#) findet und in eine schöne Webseite umwandelt. [Abbildung 3.8](#) zeigt eine solche Seite.

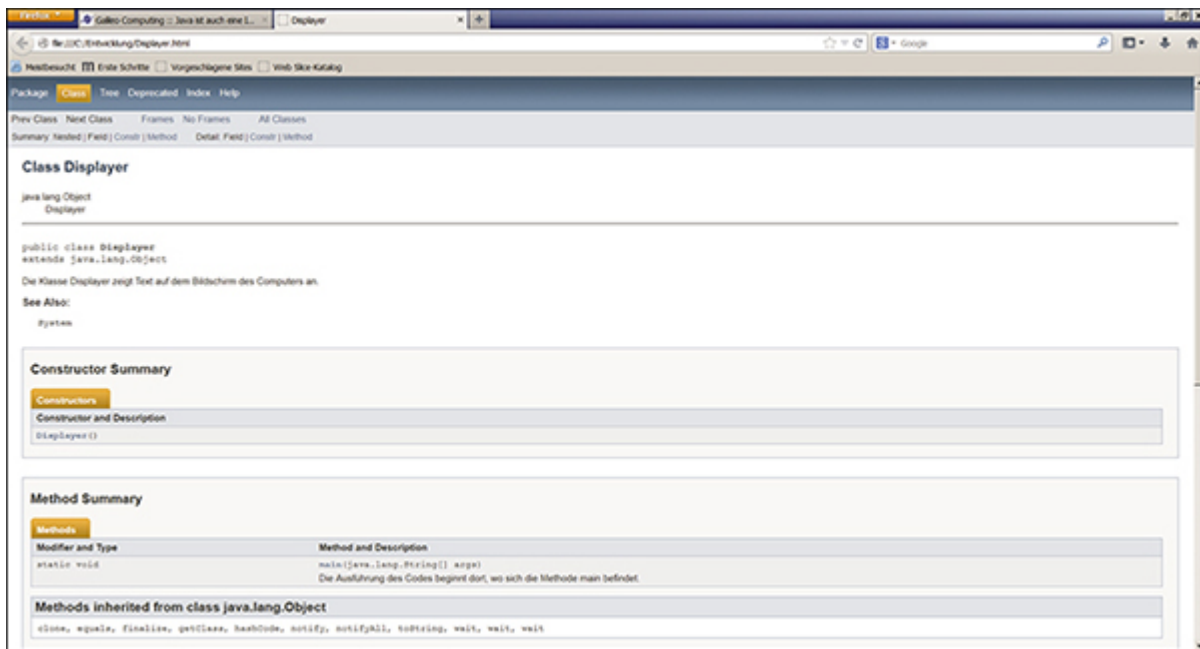


Abbildung 3.8: Die Javadoc-Seite, die vom Code in [Listing 3.6](#) generiert wurde

Javadoc-Kommentare sind eine großartige Sache. Hier ein paar der tollen Dinge, die es über sie zu berichten gibt:

- ✓ Die einzige Person, die den Java-Code zu Gesicht bekommt, ist der Programmierer, der den Code schreibt. Andere, die den Code verwenden, können herausfinden, was der Code macht, indem sie sich die automatisch erzeugte Webseite anschauen.
- ✓ Da sich andere Personen den Code nicht anschauen können, sind sie auch nicht in der Lage, Änderungen am Java-Code vorzunehmen. (Was bedeutet, dass andere keine Chance haben, in Ihren Code Fehler einzubauen.)
- ✓ Da andere nicht an den Java-Code herankommen, haben sie keine Chance, die internen Abläufe des Programms zu entschlüsseln. Alles, was diese Leute vom Code wissen müssen, ist das, was sie auf der Webseite des Codes lesen.
- ✓ Der Programmierer erstellt nicht zwei Dinge unabhängig voneinander – hier den Java-Code und dort die Dokumentation des Codes. Stattdessen erstellt der Programmierer nur einen Java-Code, in den er die Dokumentation (in Form von Javadoc-Kommentaren) einbettet.

✓ Und das Beste von allem ist, dass das Erstellen der Javadoc-Webseiten automatisiert ablaufen kann. Damit haben alle Dokumentationen das gleiche Format. Sie finden unabhängig davon, wessen Java-Code Sie verwenden, alles über den Code heraus, indem Sie eine Seite wie die in [Abbildung 3.8](#) lesen. Das klappt deshalb, weil jeder, der mit Java arbeitet, das hier verwendete Format kennt.



Sie können aus Javadoc-Kommentaren eigene Webseiten machen. Wenn Sie wissen wollen, wie das geht, besuchen Sie (die leider nur in Englisch zur Verfügung stehende Website)

www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html.

Und wie sieht Barrys Ausrede aus?

Seit Jahren erzähle ich meinen Studenten, dass sie ihren Code mit Kommentaren versehen sollen, und seit Jahren erstelle ich Code-Beispiele (wie das in [Listing 3.1](#)), die keine Kommentare enthalten. Warum?

Vier Wörter: »Kennen Sie Ihr Publikum?« Wenn Sie komplizierten, praktisch einsetzbaren Code schreiben, besteht Ihr Publikum aus anderen Programmierern, IT-Managern und Menschen, die beim Entschlüsseln dessen, was Sie gemacht haben, Hilfe benötigen. Als ich die einfachen Code-Beispiele für dieses Buch geschrieben habe, sah ich vor meinen Augen ein anderes Publikum – Sie, den Neuling als Java-Programmierer. Sie lesen keine Java-Kommentare, sondern kümmern sich in der Regel sofort um die Java-Befehle – die Befehle, die der Java-Compiler entschlüsselt. Dies ist der Grund dafür, dass es in den Listings dieses Buches so gut wie keine Kommentare gibt.

Außerdem bin ich ziemlich faul.

Kommentare verwenden, um mit dem Code zu experimentieren

Vielleicht haben Sie schon einmal gehört, dass Programmierer davon sprechen, bestimmte Teile ihres Codes *auszukommentieren*. Wenn Sie ein Programm schreiben und etwas nicht richtig funktioniert, hilft oft der Versuch, Code zu entfernen. Auf diese Weise können Sie herausfinden, was geschieht, wenn der verdächtige Code nicht mehr vorhanden ist. Das, was dann passiert, muss Ihnen auch nicht immer gefallen, weshalb Sie den Code nicht endgültig löschen sollten. Stattdessen verwandeln Sie Ihre normalen Java-Befehle in Kommentare. So könnten Sie zum Beispiel aus dem Befehl

```
System.out.println("Ich liebe Java!");
```

den Kommentar

```
// System.out.println("Ich liebe Java!");
```

machen.

Dies hindert den Java-Compiler daran, den Code zu sehen, während Sie versuchen herauszufinden, was mit Ihrem Programm nicht in Ordnung ist.

Herkömmliche Kommentare eignen sich nicht so gut dafür, Code auszukommentieren. Das Problem liegt darin, dass es nicht möglich ist, einen herkömmlichen Kommentar in einem anderen herkömmlichen Kommentar unterzubringen.

Stellen Sie sich vor, Sie wollen den folgenden Befehl auskommentieren:

```
System.out.println("Eltern,");
System.out.println("streitet nicht");
/*
 * Die Anzeige erfolgt bewusst auf vier Zeilen
 */
System.out.println("vor den");
System.out.println("Kindern!");
```

Wenn Sie versuchen, aus diesem Code einen herkömmlichen Kommentar zu machen, erhalten Sie das folgende Durcheinander:

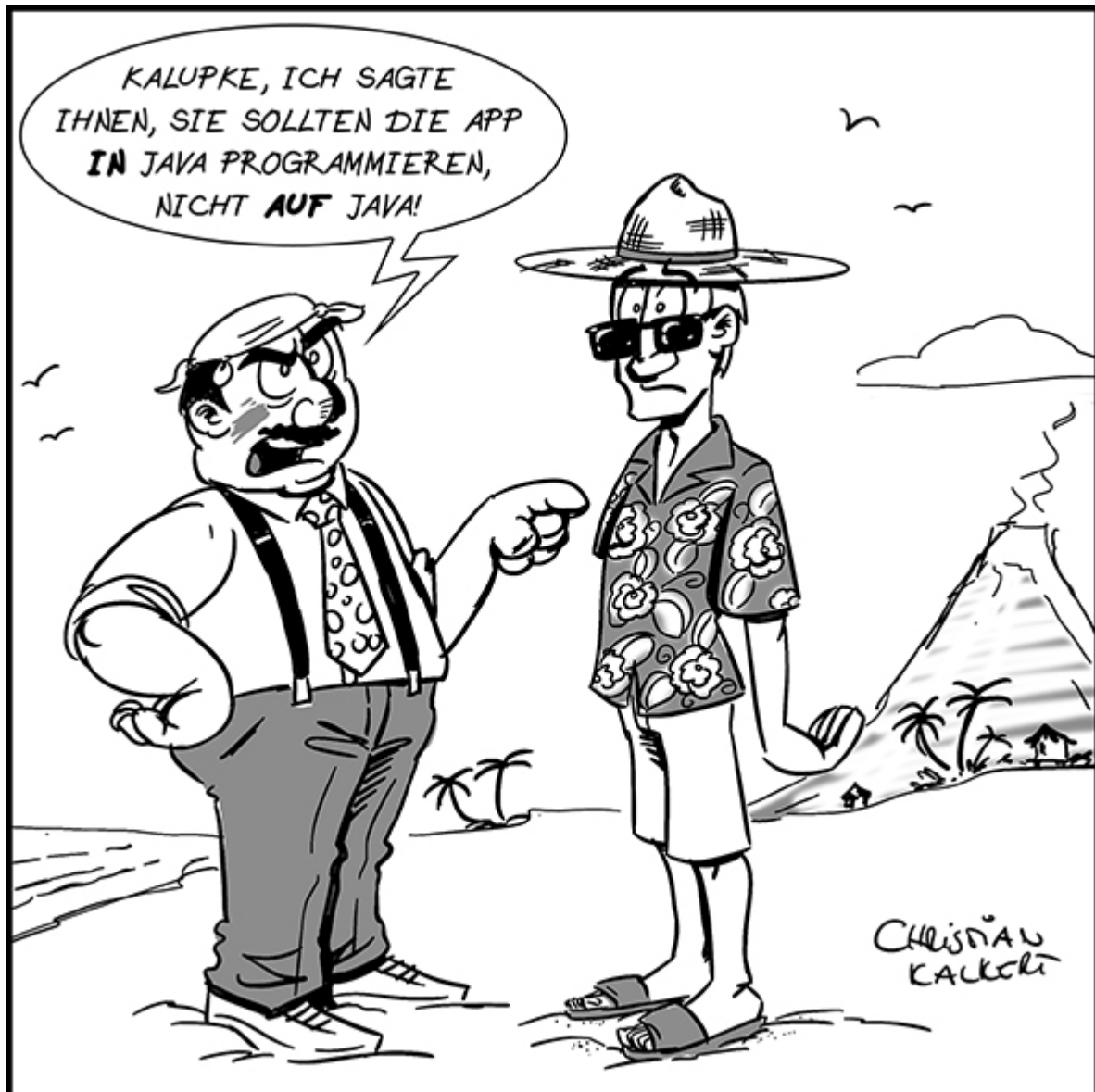
```
/*
System.out.println("Eltern,");
System.out.println("streitet nicht ");
```

```
/*  
 * Die Anzeige erfolgt bewusst auf vier Zeilen  
 */  
System.out.println("vor den");  
System.out.println("Kindern!");  
*/
```

Das erste `*/` (hinter `Die Anzeige ...`) beendet den herkömmlichen Kommentar vorzeitig. Dann werden die Befehle mit `vor den` und `Kindern` nicht auskommentiert, und das letzte `*/` bringt den Compiler um. Sie können herkömmliche Kommentare nicht ineinander verschachteln. Aus diesem Grund empfehle ich Zeilenendekommentare zu verwenden, wenn Sie mit Ihrem Code experimentieren wollen.

Teil II

Eigene Java-Programme schreiben



In diesem Teil . . .

- ✓ Sie erstellen neue Werte und ändern vorhandene.
- ✓ Sie bringen in der Logik einer Anwendung entscheidungsfällende Elemente unter.
- ✓ Sie wiederholen bei Bedarf Dinge, während das Programm läuft.

4

Das Optimum aus Variablen und ihren Werten herausholen

In diesem Kapitel

- ▶ Elementen Werte zuweisen
 - ▶ Elemente dazu bringen, bestimmte Werte aufzunehmen
 - ▶ Operatoren anwenden, um neue Werte zu erhalten
-

Die folgende Unterhaltung zwischen Mr. Charles Van Doren und Mr. Jack Barasch hat nie stattgefunden:

Charles: Eine Seescheide frisst ihr Gehirn und verwandelt sich dadurch von einem Tier in eine Pflanze.

Jack: Charles, ist das unsere endgültige Antwort?

Charles: Ja.

Jack: Wie viel Geld hast du im Moment auf dem Konto?

Charles: Ich habe auf meinem Girokonto fünfzig Euro und zweiundzwanzig Cent.

Jack: Nun, dann rufst du jetzt am besten deinen Steuerberater an, denn du hast mit deiner Antwort zur Seescheide eine Million Euro gewonnen, die bald auf deinem Girokonto landen wird. Was hältst du davon, Charles?

Charles: Das habe ich alles meiner Ehrlichkeit, meinem Fleiß und harter Arbeit zu verdanken.

Einige Punkte dieses Dialogs können mithilfe einiger Codezeilen dargestellt werden.

Eine Variable variieren

Unabhängig davon, wie Sie an Ihre Million Euro gelangen, können Sie eine Variable verwenden, um Ihren Reichtum nachzuzählen.

[Listing 4.1](#) gibt den entsprechenden Code wieder.

```
saldo = 50.22;  
saldo = saldo + 1000000.00;
```

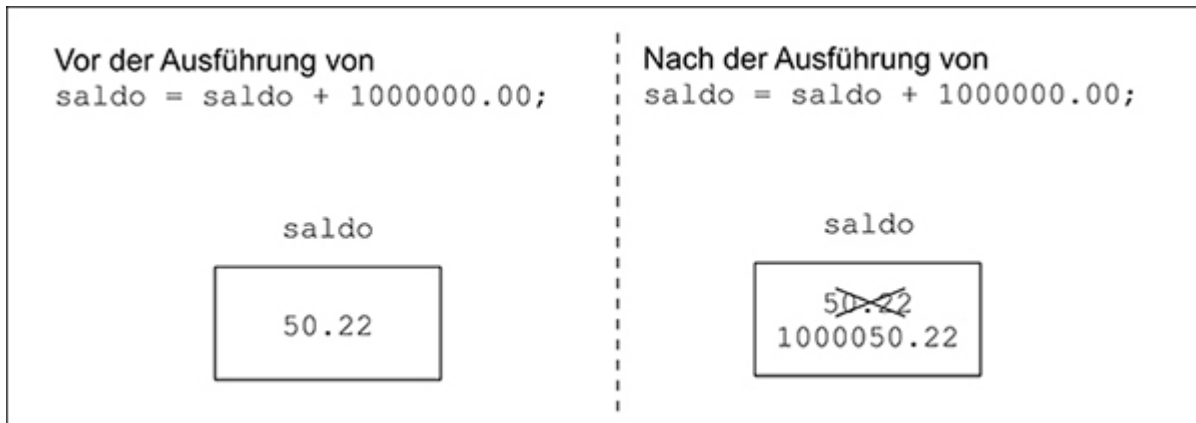
[Listing 4.1](#): Eine Variable verwenden



Beachten Sie, dass in Java das Dezimalzeichen bei der Eingabe einer Zahl über den Code ein Punkt ist.

Der Code in [Listing 4.1](#) nutzt die Variable `saldo`. Eine *Variable* ist ein Platzhalter. Sie können darin eine Zahl wie `50.22` unterbringen. Und wenn Sie in der Variablen eine Zahl platziert und Ihre Meinung geändert haben, sind Sie in der Lage, in der Variablen eine andere Zahl abzulegen. Wenn Sie in einer Variablen eine neue Zahl aufnehmen, ist natürlich die alte Zahl nicht mehr vorhanden. Wenn Sie also diese alte Zahl nicht vorher irgendwo speichern, ist sie verloren.

[Abbildung 4.1](#) zeigt den Zustand der Variablen vor und nach der Ausführung des Codes in Listing 4.1. Nachdem der erste Befehl ausgeführt worden ist, enthält die Variable `saldo` die Zahl `50.22`. Nach der Ausführung des zweiten Befehls in [Listing 4.1](#) hat sich diese Zahl plötzlich in `1000050.22` geändert. Wenn Sie an eine Variable denken, sollten Sie sich einen Ort im Arbeitsspeicher des Computers vorstellen, an dem Drähte und Transistoren `50.22`, `1000050.22` oder was sonst noch speichern. Und stellen Sie sich weiter vor, dass der Kasten im linken Teil der [Abbildung 4.1](#) von Millionen anderer Kästen dieser Art umgeben ist.



[Abbildung 4.1](#): Eine Variable (vorher und nachher)

Nun zu einigen Fachbegriffen. Das, was in einer Variablen gespeichert wird, nennt man *Wert*. Der Wert einer Variablen kann sich während der Ausführung des Programms ändern (wenn Ihnen beispielsweise Jack eine Million Mäuse gibt). Der Wert, der in einer Variablen gespeichert wird, muss nicht unbedingt eine Zahl sein. (So können Sie zum Beispiel eine Variable erstellen, die immer einen Buchstaben speichert.) Die Art des Wertes, der in einer Variablen gespeichert wird, ist der *Typ* oder *Datentyp* der Variablen.

Sie erfahren weiter hinten in diesem Kapitel im Abschnitt *Wertetypen, die Variablen annehmen können* mehr über Typen von Variablen.



Zwischen einer Variablen und einem *Variablennamen* gibt es einen kleinen, aber feinen Unterschied. Selbst wenn ich formell schreibe, verwende ich oft das Wort *Variable*, wenn *Variablennamen* gemeint ist. Genau genommen ist `saldo` ein Variablennamen, und die Variable besteht aus dem Arbeitsspeicherbereich, der `saldo` (einschließlich Typ und Wert) enthält. Wenn Sie der Meinung sind, dass die Unterscheidung zwischen *Variable* und *Variablennamen* zu viel Haarspalterei ist, willkommen im Klub.

Jeder Variablennamen ist ein Bezeichner – ein Name, den Sie in Ihrem Code selbst wählen können. So habe ich mich im Code von

[Listing 4.1](#) für den Namen `saldo` entschieden. (Wenn Sie mehr über Java-Namen wissen wollen, schauen Sie sich [Kapitel 3](#) an.)

Bevor wir [Listing 4.1](#) verlassen, sollten Sie noch einmal einen Blick auf einen nicht uninteressanten Teil des Listings werfen. Im Listing gibt es `50.22` und `1000000.00`. Jeder normale Mensch würde diese Dinge als *Zahlen* bezeichnen, aber in einem Java-Programm werden sie *Literale* genannt.

Was ist denn nun so *buchstäblich* oder *wörtlich* (das bedeutet *literal*) an `50.22` und `1000000.00`? Schauen Sie sich noch einmal die Variable `saldo` in [Listing 4.1](#) an. Sie steht zuerst für `50.22` und später für `1000050.22`. Sie könnten das Wort *Zahl* verwenden, wenn Sie von `saldo` reden. Aber in Wirklichkeit hängt das, wofür `saldo` steht, von der aktuellen Situation ab. Andererseits steht `50.22` buchstäblich (literal) für den Wert 5022/100.



Der Wert einer Variablen ändert sich, was der Wert eines Literals nicht tut.



Seit Java 7 können Sie numerischen Literalen Unterstriche hinzufügen. Statt dem guten alten `1000000.00` in [Listing 4.1](#) können Sie `saldo = saldo + 1_000_000.00` schreiben. Unglücklicherweise dürfen Sie das nicht machen, was eigentlich am natürlichsten wäre – die deutsche (1.000.000,00) oder die amerikanische Schreibweise (1,000,000.00) verwenden.

Zuweisungsbefehle

Befehle wie die in [Listing 4.1](#) werden *Zuweisungsbefehle* genannt. Sie weisen mit einem Zuweisungsbefehl einen Wert zu. Häufig ist das Ziel der Zuweisung eine Variable.

Ich empfehle Ihnen, Zuweisungsbefehle von rechts nach links zu lesen. [Abbildung 4.2](#) zeigt dies anhand der ersten Zeile von

Listing 4.1.

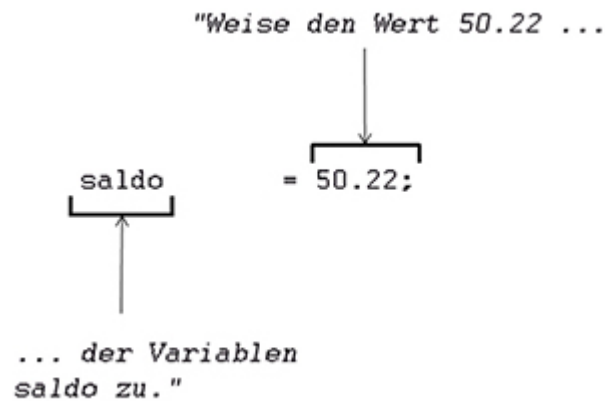


Abbildung 4.2: So sollte die erste Zeile in Listing 4.1 gelesen werden.

Die zweite Zeile in Listing 4.1 ist ein wenig komplizierter. Abbildung 4.3 stellt die Aktion dieser zweiten Zeile dar.

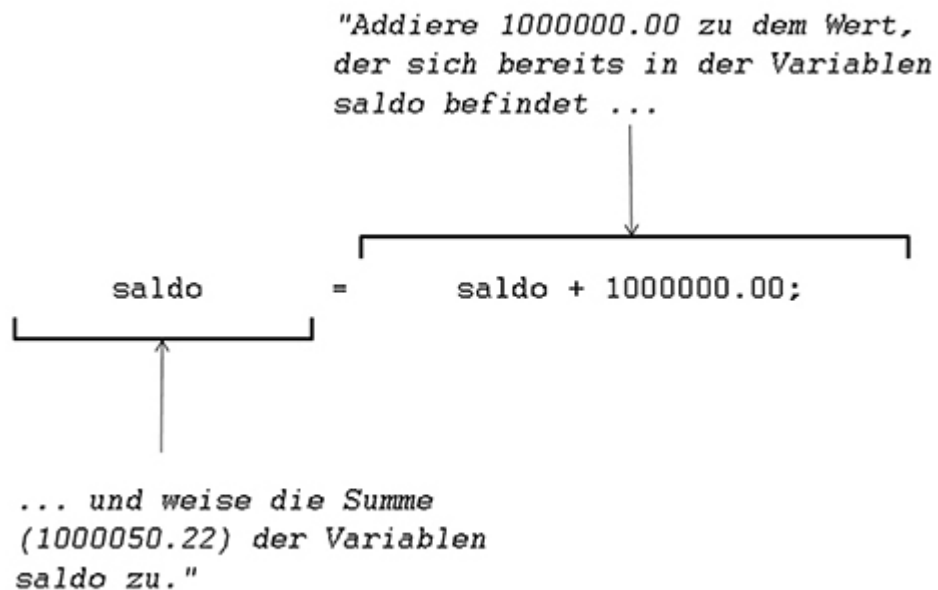


Abbildung 4.3: Die Aktion der zweiten Zeile in Listing 4.1

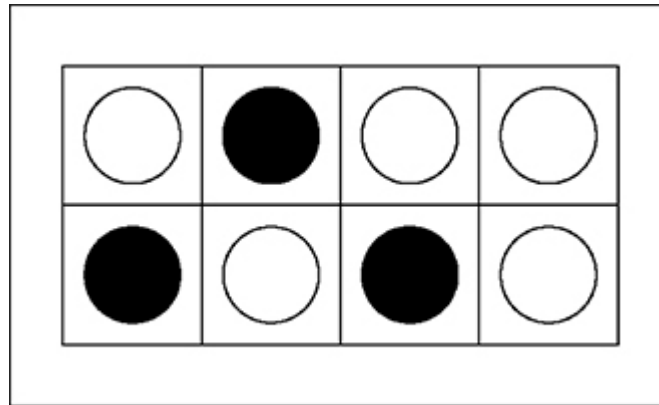


In einem Zuweisungsbefehl steht das Teil, dem ein Wert zugewiesen wird, immer auf der linken Seite des Gleichheitszeichens.

Wertetypen, die Variablen annehmen können

Haben Sie im Fernsehen die Werbung gesehen, bei der man glauben könnte, durch die Schaltkreise eines Computers zu fliegen? Nicht schlecht gemacht, oder? Diese Werbung zeigt 0en (Nullen) und 1en (Einsen), die vorbeisegeln, weil 0en und 1en das Einzige sind, mit dem Computer umgehen können. Wenn Sie glauben, dass ein Computer den Buchstaben $\mathbb{J}$ speichert, so irren Sie sich, denn in Wirklichkeit speichert er 01001010. Und anstatt den Buchstaben $\mathbb{K}$ abzulegen, speichert der Computer 01001011. Alles im Computer ist eine Folge von Nullen und Einsen. Und jeder Computerfreak weiß, dass 0 oder 1 ein *Bit* ist.

Nun steht aber 01001010 nicht nur für den Buchstaben $\mathbb{J}$, sondern auch für die Zahl 74. Dieselbe Ziffernfolge kann aber auch $1,0369608636003646 \cdot 10^{-43}$ bedeuten. Und wenn dann die Bits als Bildschirmpixel interpretiert werden, kann dieselbe Sequenz verwendet werden, um die Punkte darzustellen, die [Abbildung 4.4](#) zeigt. Die Bedeutung von 01001010 hängt also davon ab, wie die Software die Folge von Nullen und Einsen interpretiert.



[Abbildung 4.4](#): Die extreme Vergrößerung von acht schwarzen und weißen Pixeln auf dem Bildschirm

Wie können Sie nun dem Computer klarmachen, wofür 01001010 steht? Die Antwort liegt in der Idee eines *Typs*. Bei dem Typ einer Variablen handelt es sich um den Wertebereich, den die Variable speichern kann.

Ich habe die Zeilen von [Listing 4.1](#) kopiert und zu einem vollständigen Java-Programm gemacht, das [Listing 4.2](#) bildet. Wenn ich das Programm in [Listing 4.2](#) ablaufen lasse, erhalte ich auf dem Bildschirm die Ausgabe, die [Abbildung 4.5](#) zeigt.

```
public class Millionaire {
    public static void main(String args[]) {
        double saldo;
        saldo = 50.22;
        saldo = saldo + 1000000.00;
        System.out.print("Sie haben € ");
        System.out.print(saldo);
        System.out.println(" auf Ihrem Konto.");
    }
}
```

[Listing 4.2](#): Ein Programm verwendet *saldo*.

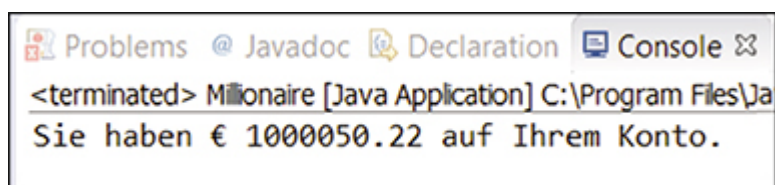


Abbildung 4.5: Das Programm in Listing 4.2 wird ausgeführt.



Die meisten Programme in diesem Buch sind textbasiert. Ein textbasiertes Programm hat keine Fenster, keine Dialogfelder, nichts dieser Art. Sie sehen immer nur Zeilen mit nacktem, unformatiertem Text. Der Benutzer gibt etwas ein, und der Computer gibt unterhalb der Eingabezeile etwas aus. So wenig Programme dieser Art auch optisch aufgepeppt sind, so enthalten sie doch die grundlegenden Konzepte der Programmierung. Außerdem lassen sie sich von einem Programmierneuling leichter lesen, schreiben und verstehen. Aus diesem Grund fahre ich in diesem Buch zweigleisig. Die Beispiele im gedruckten Buch sind fast alle textbasiert, während Sie auf der amerikanischen Webseite dieses Buches (allmycode.com/JavaForDummies) aufregendere Versionen der meisten Beispiele mit Fenstern, Schaltflächen, Textfeldern und anderen Elementen vorfinden. Denken Sie aber daran, dass diese Webseite nur englischsprachig ist.

In Listing 4.2 lautet die erste Zeile im Körper der Methode `main`

```
double saldo;
```

Diese Zeile wird *Variablendeklaration* genannt. Wenn Sie diese Zeile in Ihr Programm aufnehmen, ist das so, als wenn Sie Ihre Absicht bekannt geben, im Programm eine Variable mit dem Namen `saldo` einzubinden.

In dieser Variablendeklaration ist das Wort `double` ein Java-Schlüsselwort. Es sagt dem Computer, welche Art von Werten Sie in der Variablen `saldo` speichern wollen. Insbesondere steht das Wort `double` für Zahlen zwischen $-1,8 * 10^{308}$ und $1,8 * 10^{308}$. (Das sind riesengroße Zahlen mit 308 Nullen vor dem Dezimalzeichen. Nur die reichsten Menschen dieser Erde würden Schecks mit 308 Nullen ausstellen. Die zweite dieser Zahlen bedeutet eins-komma-acht Gazazzo-Zillion-Kaskillionen. Die Zahl $1,8 * 10^{308}$ ist eine Konstante, die vom *International Bureau of Weights and Measures* definiert wurde und die Zahl der

exzentrischen Programmierer zwischen Sunnyvale, Kalifornien, und der Andromeda-Galaxie M31 angibt.)

Wichtiger noch als der riesige Wertebereich des Schlüsselwortes `double` ist die Tatsache, dass `double`-Zahlen auch Nachkommastellen haben können. Nachdem Sie `saldo` als `double`-Typ deklariert haben, können Sie in dieser Variablen beliebige Zahlen wie 50.22, 0.02398479 oder –3.0 speichern. Wenn ich `saldo` nicht vom Typ `double` deklariert hätte, wäre ich vielleicht nur in der Lage gewesen, die gute alte Ganzzahl 50 abzuspeichern.

Nachkommastellen

Java kennt zwei verschiedene Typen mit Nachkommastellen: `double` und `float`. Worin besteht der Unterschied? Wenn Sie eine Variable vom Typ `double` deklarieren, reserviert der Computer für den Inhalt der Variablen 64 Bits im Arbeitsspeicher. Bei Variablen vom Typ `float` reserviert der Computer nur 32 Bits.

Sie können [Listing 4.1](#) ändern und `saldo` als Variablentyp `float` deklarieren:

```
float saldo
```

Natürlich reichen 32 Bits aus, um eine kleine Zahl wie 50.22 aufzunehmen, oder? Nun, sie reichen aus – und doch wieder nicht. Sie können mit 32 Bits problemlos 50.00 speichern. Es würden sogar sechs Bits ausreichen, um 50.00 zu speichern. Aber es ist nicht die Größe der Zahl entscheidend. Hier geht es um die Genauigkeit. Bei einer 64-Bit-Variablen vom Typ `double` wird der größte Teil der Bits für die Nachkommastellen verwendet. Um den .22-Teil von 50.22 zu speichern, benötigen Sie mehr als die mageren 32 Bits, die der Typ `float` bereitstellt.

Glauben Sie wirklich, was Sie gerade gelesen haben – dass Sie mehr als 32 Bits benötigen, um .22 zu speichern? Um Sie davon zu überzeugen, dass dies wirklich stimmt, habe ich in [Listing 4.2](#) einige Änderungen am Code vorgenommen. Ich habe `saldo` als Typ `float` deklariert und die ersten drei Befehle in der Methode `main` wie folgt geändert:

```
float saldo;  
saldo = 50.22F;  
saldo = saldo + 1000000.00F
```

(Wenn Sie wissen wollen, warum ich bei 50.22F und 1000000.00F den Buchstaben `F` verwende, siehe [Tabelle 4.1](#).) Ich erhielt daraufhin diese

Ausgabe:

Sie haben € 1000050.25 auf Ihrem Konto.

Vergleichen Sie dies mit der Ausgabe in [Abbildung 4.5](#). Wenn ich vom Typ `double` zum Typ `float` wechsele, erhält Charles auf seinem Konto einen Bonus von drei Cent gutgeschrieben. Durch den Wechsel zum 32-Bit-Typ `float` hat die Genauigkeit an der zweiten Nachkommastelle der Variablen `saldo` gelitten. Das ist nicht gut.

Ein weiteres Problem mit `float`-Werten ist kosmetischer Natur. Schauen Sie sich in [Listing 4.2](#) noch einmal die Literale `50.22` und `1000000.00` an. Die Gesetze Javas verlangen, dass Literale wie diese jeweils 64 Bits belegen. Dies bedeutet, dass Sie Schwierigkeiten bekommen werden, wenn Sie `saldo` als `float`-Typ deklarieren. Sie stehen dann nämlich vor dem Problem, diese 64-Bit-Literale in der nur 32 Bits großen `saldo`-Variablen unterzubringen. Zum Ausgleich können Sie von `double`- zu `float`-Literalen wechseln, indem Sie an jedes `double`-Literal ein `F` hängen, wobei eine Zahl mit einem `F` am Ende komisch aussieht.

Wenn Sie ein wenig mit Zahlen herumspielen wollen, sollten Sie <http://babbage.cs.qc.edu/IEEE-754> besuchen. Diese englischsprachige Seite nimmt sich jede Zahl vor, die Sie eingeben, und zeigt Ihnen, wie die Zahl in der 32-Bit- und in der 64-Bit-Version dargestellt wird.



Der Eine-Million-Euro-Jackpot in [Listing 4.2](#) ist ziemlich beeindruckend. Aber dieses Listing stellt nicht den besten Weg dar, um mit Euro-Beträgen umzugehen. Um in einem Java-Programm Währungen optimal darzustellen, sollten Sie die Typen `double` und `float` meiden und sich stattdessen für einen Datentyp namens `BigDecimal` entscheiden.

Text anzeigen

Die letzten drei Befehle in [Listing 4.2](#) verwenden einen netten Formatierungstrick. Sie möchten auf dem Bildschirm auf einer einzelnen Zeile verschiedene Elemente anzeigen. Sie packen diese Dinge in separate Anweisungen. Mit Ausnahme des letzten Befehls rufen alle Anweisungen `System.out.print` auf. (Beim letzten Befehl handelt es sich um einen Aufruf von

`System.out.println`.) Aufrufe von `System.out.print` zeigen Text auf einer Zeile an und belassen den Cursor auf dieser Zeile. Da sich nun der Cursor nach dem Aufruf von `System.out.print` noch in derselben Zeile wie der vorherige Text befindet, setzt der nächste Aufruf von `System.out.wasauchimmer` die Ausgabe auf eben dieser Zeile fort. Wenn Sie mehrere Aufrufe von `print` mit einem einzelnen Aufruf von `println` abschließen, erhalten Sie als Ergebnis eine ansprechende Ausgabe wie die in [Abbildung 4.5](#).



Ein Aufruf von `System.out.print` gibt etwas aus und sorgt dafür, dass der Cursor am Ende der Ausgabe auf der entsprechenden Zeile wartet. Ein Aufruf von `System.out.println` gibt etwas aus und beendet seinen Job damit, dass der Cursor an den Anfang einer neuen Zeile gesetzt wird.

Ganzzahlen

»1995 hatte die amerikanische Durchschnittsfamilie 2,3 Kinder.«

An dieser Stelle gibt es immer einen Schlaumeier, der bemerkt, dass im wahren Leben keine Familie genau 2,3 Kinder haben kann. Ganze Zahlen spielen eben in unserem Leben eine wichtige Rolle. Deshalb können Sie in Java auch eine Variable deklarieren, in der sich nichts anderes als Ganzzahlen ablegen lassen. [Listing 4.3](#) zeigt ein Programm, das ganzzahlige Variablen nutzt.

```
public class ElevatorFitter {
    public static void main(String args[]) {
        int weightOfAPerson;
        int elevatorWeightLimit;
        int numberOfPeople;
        weightOfAPerson = 150;
        elevatorWeightLimit = 1400;
        numberOfPeople =
            elevatorWeightLimit / weightOfAPerson;
        System.out.print("Es passen ");
        System.out.print(numberOfPeople);
    }
}
```



```
System.out.println(" Personen in den Fahrstuhl.");  
    }  
}
```

[Listing 4.3](#): Den Typ *int* verwenden

Hinter dem Programm in [Listing 4.3](#) verbirgt sich diese Geschichte:

In einem Hotel gibt es einen Fahrstuhl, dessen Kapazität (`elevatorWeightLimit`) auf 1400 Pfund begrenzt ist. An einem Wochenende findet im Hotel das Familientreffen der Brickenchickers statt. Ein Zweig der Familie war mit eineiigen Zehnlingen gesegnet (zehn Geschwister mit derselben Statur). Normalerweise wiegt jeder der Brickenchicker-Zehnlinge genau 145 Pfund. Aber am Samstag gab es mittags ein großartiges Büffet, zu dem auch Erdbeertörtchen gehörten. Das führte dazu, dass jeder der Brickenchicker-Zehnlinge nun 150 Pfund wog. Unmittelbar nach dem Essen trafen sich alle zehn Brickenchickers vor dem Fahrstuhl. (Warum auch nicht? Sie hatten fast immer die gleichen Gedankengänge.) Es stellte sich nun die Frage, wie viele der Zehnlinge in den Fahrstuhl passen. Denken Sie daran, dass das Kabel des Fahrstuhls reißt, wenn Sie ein Gramm mehr als die zulässigen 1400 Pfund in den Fahrstuhl packen, und dieses Reißen würde die Brickenchickers in einen schrecklichen Tod stürzen lassen.

Die Antwort auf die Frage (die Ausgabe des Programms in [Listing 4.3](#)) zeigt [Abbildung 4.6](#).



[Abbildung 4.6](#): Retten Sie die Brickenchickers. Retten Sie die Welt.

Im Zentrum des Aufzugsproblems der Familie Brickenchicker stehen Ganzzahlen – Zahlen ohne Dezimalstellen. Wenn Sie

1.400 durch 150 teilen, erhalten Sie 9 1/3, wobei Sie 1/3 nicht ernst nehmen sollten. 1/3 Mensch lässt sich nur schlecht in einem Fahrstuhl unterbringen. Diese Tatsache berücksichtigt auch Java. In [Listing 4.3](#) sind alle drei Variablen – `weightOfAPerson` (Gewicht einer Person), `elevatorWeightLimit` (Gewichtsbeschränkung des Fahrstuhls) und `numberOfPeople` (Anzahl Personen) – vom Typ `int`. Bei einem `int`-Wert handelt es sich um eine Ganzzahl. Wenn Sie einen `int`-Wert durch einen anderen `int`-Wert teilen (was der Schrägstrich in [Listing 4.3](#) bewirkt), erhalten Sie immer eine Ganzzahl als Ergebnis. Wenn Sie 1.400 durch 150 teilen, erhalten Sie 9, nicht 9 1/3. Sie sehen dies in [Abbildung 4.6](#). Kurz gefasst, die folgenden Befehle sorgen dafür, dass auf dem Bildschirm 9 angezeigt wird:

```
numberOfPeople =  
    elevatorWeightLimit / weightOfAPerson;  
System.out.print(numberOfPeople);
```

Deklarationen kombinieren und Variablen initialisieren

Werfen Sie noch einmal einen Blick auf Listing 4.3. In diesem Listing gibt es drei Variablendeklarationen – eine für jede der drei im Programm vorkommenden `int`-Variablen. Ich hätte dasselbe Ergebnis auch mit nur einer Deklaration erreichen können:

```
int weightOfAPerson, elevatorWeightLimit, numberOfPeople;
```



Wenn sich zwei Variablen im Typ unterscheiden, können Sie sie nicht in derselben Deklaration anlegen. Wenn Sie zum Beispiel eine `int`-Variable mit dem Namen `gewichtVonFred` und eine `double`-Variable mit dem Namen `saldoVonFredsKonto` benötigen, müssen Sie zwei eigenständige Variablendeklarationen verwenden.

Sie können Variablen in der Deklaration einen Anfangswert mitgeben. In [Listing 4.3](#) kann eine Deklaration mehrere Zeilen der

Methode `main` ersetzen (alle bis auf die Aufrufe von `print` und `println`).

Vier Verfahren, um Ganzzahlen zu speichern

Java kennt vier ganzzahlige Datentypen. Diese sind `byte`, `short`, `int` und `long`. Anders als das Problem mit der Genauigkeit bei den Typen `double` und `float` geht es bei Ihrer Wahl des ganzzahligen Datentyps ausschließlich um die Größe der Zahlen, die Sie in einer solchen Variablen speichern wollen. Wenn Sie vorhaben, Zahlen abzulegen, die größer als 127 sind, können Sie `byte` nicht verwenden, und bei Zahlen, die größer als 32.767 sind, müssen Sie auf `short` verzichten.

Meistens werden Sie `int` verwenden. Wenn Sie aber Zahlen speichern müssen, die größer als 2.147.483.647 sind, bleibt Ihnen nichts anderes übrig, als `int` durch `long` zu ersetzen. (Die Obergrenze einer Zahl vom Typ `long` liegt bei 9.223.372.036.854.775.807.) Alle Einzelheiten zu den Datentypen enthält [Tabelle 4.1](#).

```
int weightOfAPerson = 150, elevatorWeightLimit = 1400
',
    numberOfPeople = elevatorWeightLimit/weightOfAPerson;
n;
```

Wenn Sie so etwas machen, sprechen Sie nicht davon, dass Sie Variablen Werte *zuweisen*. Die Teile der Deklarationen mit den Gleichheitszeichen werden nicht wirklich Zuweisungsbefehl genannt. Stattdessen heißt es bei Variablen, dass Sie sie *initialisieren*. Ob Sie es mir nun glauben oder nicht, aber sich das zu merken, kann sehr hilfreich sein.

Das Initialisieren von Variablen hat – wie alles im Leben – Vor- und Nachteile:

- ✓ **Wenn Sie sechs Zeilen von [Listing 4.3](#) in einer Deklaration komprimieren, fassen Sie den Code naturgemäß zusammen.** Manchmal lässt sich zusammengefasster Code leichter lesen. Manchmal aber auch nicht. Die Beurteilung obliegt Ihnen als Programmierer.
- ✓ **Indem Sie eine Variable initialisieren, vermeiden Sie automatisch, dass es zu bestimmten**

Programmierungsfehlern kommt. Sie finden hierzu ein Beispiel in [Kapitel 7](#).

- ✓ **Unter Umständen haben Sie keine Wahl. Die Natur Ihres Codes zwingt Sie dazu, eine Variable zu initialisieren oder darauf zu verzichten.** Ein Beispiel, das ein Initialisieren einer Variablen nicht zulässt, finden Sie in [Kapitel 6](#).

Die Atome: Javas primitive Typen

Die Wörter *int* und *double*, die ich in den vorherigen Abschnitten beschreibe, sind in Java Beispiele für *primitive Typen* (die auch *einfache Typen* genannt werden). Die Sprache Java kennt genau acht primitive Typen. Da Sie gerade erst mit Java anfangen, können Sie alle Typen bis auf vier ignorieren. [Tabelle 4.1](#) enthält eine Liste aller primitiven Typen.

Name des Typs	Wie ein Literal aussieht	Wertebereich
Ganzzahlige Typen		
byte	(byte) 42	−128 bis 127
short	(short) 42	−32.768 bis 32.767
int	42	−2.147.483.648 bis 2.147.483.647
long	42L	−9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
Dezimalzahlige Typen		
float	42.0F	−3,4 * 10 ³⁸ bis 3,4 * 10 ³⁸
double	42.0	−1,8 * 10 ³⁰⁸ bis 1.8 * 10 ³⁰⁸
Aus Zeichen bestehende Typen		
char	‘A’	Tausende von Zeichen
Logische Typen		
boolean	true	wahr; falsch

Tabelle 4.1: Javas primitive Typen

Die Typen, die Sie nicht ignorieren sollten, sind `int`, `double`, `char` und `boolean`. Ich gehe weiter vorn in diesem Kapitel auf die Typen `double` und `int` ein, während sich die nächsten Abschnitte mit `char` und `boolean` beschäftigen.

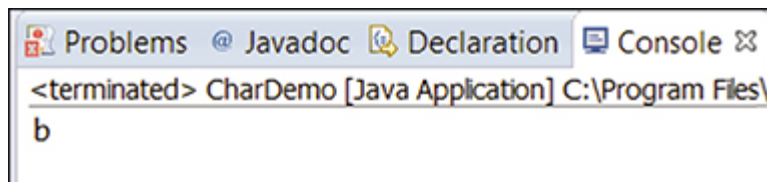
Der Typ »char«

Vor noch nicht allzu langer Zeit dachten viele, dass sich Computer nur mit dem Berechnen riesiger Zahlenmengen beschäftigen. Heute, wo es Textverarbeitungen gibt, denkt niemand mehr so. Wenn Sie sich also in den letzten 30 Jahren tiefgefroren in einer Kältekammer aufgehalten haben, wissen Sie jetzt, dass Computer auch Buchstaben, Interpunktionszeichen und anderes abspeichern.

Der Java-Typ, der verwendet wird, um Zeichen zu speichern, wird *char* genannt. (Dieser Begriff leitet sich vom englischen Ausdruck für Zeichen, *Character*, ab.) [Listing 4.4](#) enthält ein einfaches Programm, das den Typ `char` verwendet. [Abbildung 4.7](#) zeigt, was das Programm in [Listing 4.4](#) ausgibt.

```
public class CharDemo {  
    public static void main(String args[]) {  
        char myLittleChar = 'b';  
        char myBigChar = Character.toUpperCase(myLittleC  
har);  
        System.out.println(myBigChar);  
    }  
}
```

[Listing 4.4](#): Den Typ *char* verwenden



[Abbildung 4.7](#): Das Ergebnis der Ausführung des Programms in Listing 4.4, wie es in der Eclipse View »Console« wiedergegeben wird

In [Listing 4.4](#) speichert die erste Initialisierung den Buchstaben `b` in der Variablen `myLittle Char`. Beachten Sie, dass `b` bei der Initialisierung von einfachen Anführungszeichen eingeschlossen wird. In Java beginnen und enden `char`-Litereale mit einem einfachen Anführungszeichen.



In einem Java-Programm schließen einfache Anführungszeichen den Buchstaben in einem `char`-Literal ein.

Wenn Sie Probleme damit haben, die Begriffe *Zuweisung*, *Deklaration* und *Initialisierung* einzusortieren, schauen Sie sich weiter vorn in diesem Kapitel den Abschnitt *Deklarationen kombinieren und Variablen initialisieren* an.

Bei der zweiten Initialisierung in [Listing 4.4](#) ruft das Programm eine API-Methode auf, die `Character.toUpperCase` heißt. (Sie erfahren in [Kapitel 3](#) mehr über das Java Application Programming Interface, abgekürzt *API*.) Die Methode `Character.toUpperCase` macht genau das, was ihr englischer Name vermuten lässt – sie wandelt einen Kleinbuchstaben (den Buchstaben `b`) in einen Großbuchstaben (den Buchstaben `B`) um. Dieser Großbuchstabe wird dann der Variablen `myBigChar` zugewiesen und anschließend auf dem Bildschirm ausgegeben.

Wenn es Sie lockt, den folgenden Code zu schreiben:

```
//Nicht nachmachen
```

```
Character.toUpperCase = 'barry';
```

sollten Sie der Versuchung widerstehen. Sie können in einer `char`-Variablen immer nur einen einzigen Buchstaben speichern, und Sie dürfen zwischen einfachen Anführungszeichen nie mehr als einen Buchstaben unterbringen. Wenn Sie Wörter oder Sätze (und nicht nur einen einzelnen Buchstaben) speichern wollen, benötigen Sie etwas, das `String` genannt wird. (Sie finden weiter hinten in diesem Kapitel im Abschnitt *Die Moleküle und Verbindungen: die Referenztypen* mehr über den `String`-Typ heraus.)



Wenn Sie Programme in anderen Sprachen schreiben, kennen Sie vielleicht etwas, das sich ASCII-Zeichenkodierung nennt. Die meisten Sprachen verwenden ASCII; Java verwendet Unicode. Bei der alten ASCII-Darstellung standen jedem Zeichen nur 8 Bits zu, während Unicode-Zeichen 8, 16 oder 32 Bits einnehmen. Während ASCII nur die Zeichen des bekannten lateinischen (englischen) Alphabets speichert, hat Unicode Raum für die Zeichen fast aller auf der Welt gesprochenen Sprachen. Das einzige Problem besteht darin, dass einige der Methoden der Java-API nur auf 16-Bit-Unicode abgestimmt sind. Wenn Sie zum Beispiel eine Methode verwenden, um auf dem Bildschirm `Hallo` zu schreiben, was dort aber zu `H a l l o` wird, müssen Sie sich mit der Dokumentation zu dieser Methode beschäftigen, um herauszubekommen, ob in der Dokumentation etwas zum Umgang mit Unicode steht.

Sie sollten beachten, dass die beiden Methoden `Character.toUpperCase` und `System.out.println` in [Listing 4.4](#) ganz unterschiedlich verwendet werden. Die Methode `Character.toUpperCase` wird als Teil einer Initialisierung oder als Zuweisungsbefehl aufgerufen, während die Methode `System.out.println` für sich steht. Sie können in [Kapitel 7](#) mehr darüber herausfinden.

Der Typ »boolean«

Eine Variable vom Typ `boolean` speichert einen von zwei Werten – `true` (wahr) oder `false` (falsch). [Listing 4.5](#) demonstriert die Verwendung einer `boolean`-Variablen und [Abbildung 4.8](#), was das Programm in [Listing 4.5](#) ausgibt.

```
public class ElevatorFitter2 {
    public static void main(String args[]) {
        System.out.println("Wahr oder Falsch?");
        System.out.println("Es passen alle zehn der "
);
        System.out.println("Brickenchicker-
Zehnlinge ");
```



```

        System.out.println("in den Fahrstuhl:");
        System.out.println();
        int weightOfAPerson = 150;
        int elevatorWeightLimit = 1400;
        int numberOfPeople =
            elevatorWeightLimit / weightOfAPerson;

        boolean allTenOkay = numberOfPeople >= 10;
        System.out.println(allTenOkay);
    }
}

```

[Listing 4.5](#): Den Typ *boolean* verwenden

In [Listing 4.5](#) ist die Variable `allTenOkay` vom Typ `boolean`. Um für diese Variable einen Wert zu finden, prüft das Programm nach, ob `numberOfPeople` größer oder gleich zehn ist. (Das Symbol `>=` steht für *größer oder gleich*.)



[Abbildung 4.8](#): Die Brickenchicker-Zehnlinge sind wieder da.

An dieser Stelle zählt es sich aus, es mit der Terminologie genau zu nehmen. Jede Komponente eines Java-Programms, die einen Wert hat, wird als *Ausdruck* bezeichnet.

Wenn Sie schreiben

```
weightOfAPerson = 150;
```

ist `150` ein Ausdruck (ein Ausdruck, dessen Wert `150` beträgt).

Und wenn Sie schreiben

```
numberOfEggs = 2 + 2;
```

ist `2 + 2` ein Ausdruck (da `2 + 2` den Wert `4` hat). Wenn Sie schreiben


```
int numberOfPeople =  
    elevatorWeightLimit / weightOfAPerson;  
dann ist elevatorWeightLimit / weightOfAPerson ein  
Ausdruck. (Der Wert dieses Ausdrucks hängt davon ab, welche  
Werte die Variablen elevatorWeightLimit und  
weightOfAPerson haben, wenn der Code mit dem Ausdruck  
ausgeführt wird.)
```



Jede Komponente eines Java-Programms, die einen Wert hat, ist ein *Ausdruck*.

In [Listing 4.5](#) ist der Code `numberOfPeople >= 10` ein Ausdruck. Der Wert dieses Ausdrucks hängt von dem Wert ab, der in der Variablen `numberOfPeople` gespeichert ist. Aber Sie wissen aus der Vorgeschichte, dass der Wert von `numberOfPeople` nicht größer als zehn werden kann. Bei den vorgegebenen Gewichten und der Gewichtsbeschränkung des Fahrstuhls führt die Berechnung von `numberOfPeople` dazu, dass `numberOfPeople >= 10` den Wert `false` erhält, was wiederum dazu führt, dass auch der Variablen `allTenOkay` der Wert `false` zugewiesen wird.



Ich rufe in [Listing 4.5](#) `System.out.println()` mit leeren Klammern auf. Dadurch erreiche ich, dass das Programm bei der Ausgabe eine leere Zeile einfügt.

Die Moleküle und Verbindungen: die Referenztypen

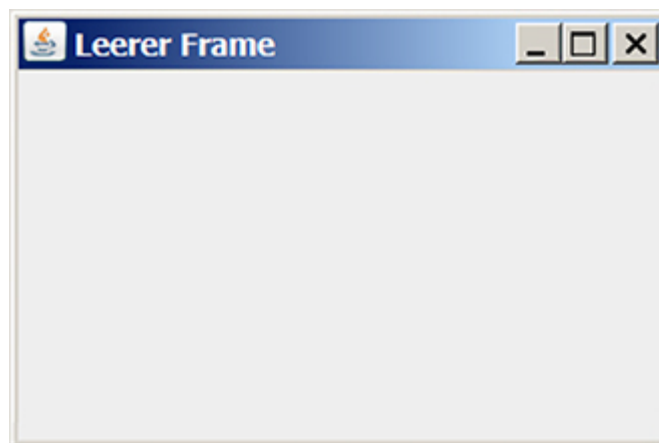
Wenn Sie einfache Elemente miteinander verbinden, erhalten Sie kompliziertere Elemente. So ist das eigentlich immer. Nehmen Sie primitive Java-Typen, rühren Sie sie zu einem »primitiven« Eintopf

untereinander, und was erhalten Sie? Einen komplexeren Typ, der *Referenztyp* genannt wird.

Das Programm in [Listing 4.6](#) verwendet Referenztypen. [Abbildung 4.9](#) zeigt, was geschieht, wenn Sie dieses Programm ausführen.

```
import javax.swing.JFrame;
public class ShowAFrame {
    public static void main(String args[]) {
        JFrame myFrame = new JFrame();
        String myTitle = "Leerer Frame";
        myFrame.setTitle(myTitle);
        myFrame.setSize(300, 200);
        myFrame.setDefaultCloseOperation
            (JFrame.EXIT_ON_CLOSE);
        myFrame.setVisible(true);
    }
}
```

[Listing 4.6](#): Referenztypen verwenden



[Abbildung 4.9](#): Ein leerer Frame

Das Programm in [Listing 4.6](#) verwendet zwei Referenztypen. Beide sind bereits in der Java-API definiert worden. Einer dieser Typen (der, den Sie fast immer verwenden werden) heißt `String`. Der andere Typ (der, den Sie verwenden können, um GUIs zu erstellen) heißt `JFrame`.

Ein `String` besteht aus Zeichen. Das ist so, als wenn es in einer Zeile mehrere `char`-Werte gäbe. Dadurch, dass die Variable `myTitle` als `String`-Variable deklariert wird, kann ihr der Text "Leerer Frame" zugewiesen werden, der dann für den neuen Frame als Titel dient. Die Klasse `String` wird in der Java-API deklariert.



In Java werden die Buchstaben in einem `String`-Literal von doppelten Anführungszeichen eingeschlossen.

Ein `JFrame` entspricht in Java quasi einem Fenster. (Der einzige Unterschied besteht darin, dass es `JFrame` und nicht `Window` genannt wird.) Damit [Listing 4.6](#) kurz und knapp bleibt, habe ich mich entschlossen, meinen Frame leer zu lassen – keine Schaltflächen, keine Felder, nichts.

Selbst bei einem vollständig leeren Frame muss [Listing 4.6](#) auf Tricks zurückgreifen, die ich erst weiter hinten in diesem Buch beschreibe. Versuchen Sie also gar nicht erst, jedes Wort dieses Listings zu verstehen. Was Sie aus [Listing 4.6](#) aber unbedingt mitnehmen sollten, sind die beiden Variablendeklarationen. Beim Schreiben des Programms habe ich die beiden Namen `myTitle` und `myFrame` erfunden und die entsprechenden Typen (`String` beziehungsweise `JFrame`) zugewiesen.

Sie können `String` und `JFrame` in Javas API-Dokumentation nachschlagen. Aber bevor Sie das tun, lassen Sie mich erklären, was Sie dort finden. Sie erfahren dort, dass `String` und `JFrame` die Namen von Java-Klassen sind. Klassen sind nichts anderes als die Namen von Referenztypen. (Informationen darüber, was Java-Klassen sind, stehen im Abschnitt über die objektorientierte Programmierung – OOP – in [Kapitel 1](#).)

Sie können `saldo` für `double`-Variablen reservieren, indem Sie

```
double saldo;
```

oder

```
double saldo = 50.22;
```

schreiben. Und so können Sie auch `myFrame` für eine `JFrame`-Variable reservieren:

```
JFrame myFrame;
```

oder

```
JFrame myFrame = new JFrame();
```



Jede Java-Klasse ist ein Referenztyp. Wenn Sie eine Variable so deklarieren, dass sie etwas enthält, das kein primitiver Typ ist, wird der Name der Variablen (fast immer) zum Namen einer Java-Klasse.

Wenn Sie eine Variable so deklarieren, dass sie vom Typ `int` ist, können Sie sich sicherlich vorstellen, was dies für den Computer bedeutet: Irgendwo in seinem Arbeitsspeicher wird für den Wert der Variablen ein Platz reserviert. Dieser Speicherort enthält eine Folge von Bits, deren Anordnung dafür sorgt, dass eine bestimmte Ganzzahl wiedergegeben wird.

Diese Erklärung eignet sich gut für primitive Typen wie `int` oder `double`, aber wie sieht das bei Variablen aus, die Sie als Referenztyp deklariert haben? Wie sieht das bei einer Variablen wie `myFrame` aus, die vom Typ `JFrame` ist?

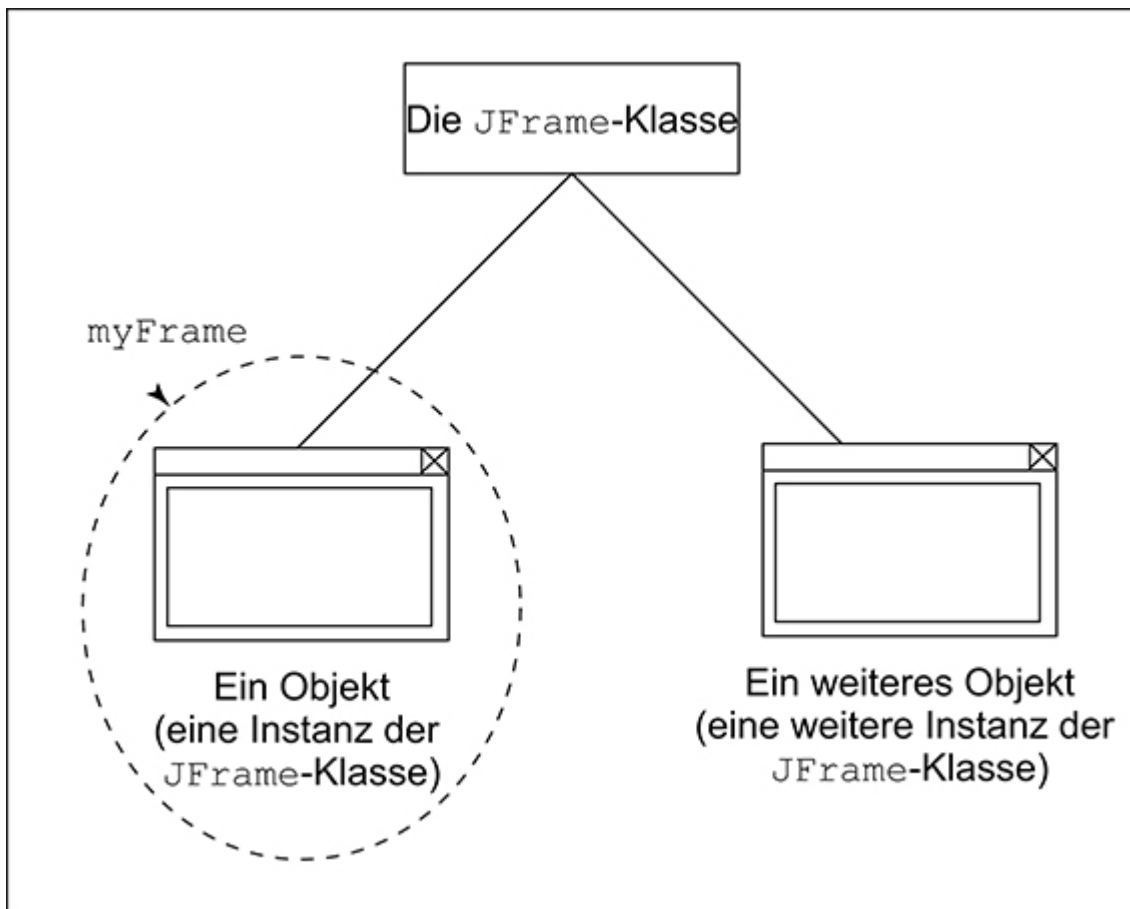
Nun, was bedeutet es, *DerTaucher* als `F` zu deklarieren? Ein Gedicht von Friedrich von Schiller? Was würde es bedeuten, die folgende Deklaration zu schreiben?

```
FvSchillerGedicht derTaucher;
```

Das bedeutet, dass eine Klasse von Elementen als `FvSchillerGedicht` bezeichnet wird, und dass `derTaucher` auf eine Instanz dieser Klasse verweist. Oder mit anderen Worten, `derTaucher` ist ein Objekt, das zur Klasse `FvSchillerGedicht` gehört.

Da `JFrame` eine Klasse ist, können Sie Objekte dieser Klasse erstellen (siehe [Kapitel 1](#)). Jedes Objekt (jede Instanz der Klasse `JFrame`) ist ein echter Frame – ein Fenster, das auf dem Bildschirm erscheint, wenn Sie den Code in [Listing 4.6](#) ausführen. Indem Sie in Ihrer Deklaration festlegen, dass die Variable

`myFrame` vom Typ `JFrame` ist, reservieren Sie die Verwendung des Namens `myFrame`. Diese Reservierung sagt dem Computer, dass `myFrame` auf ein tatsächlich existierendes Objekt vom Typ `JFrame` verweisen kann. Oder anders ausgedrückt: `myFrame` kann zu einem Kurznamen für eines der Fenster werden, die auf dem Bildschirm des Computers erscheinen. [Abbildung 4.10](#) stellt diese Situation dar.



[Abbildung 4.10](#): Die Variable `myFrame` verweist auf eine Instanz der Klasse `JFrame`.



Wenn Sie `KlassenName variablenName` deklarieren, sagen Sie damit aus, dass eine bestimmte Variable auf eine Instanz einer bestimmten Klasse verweisen kann.



In [Listing 4.6](#) reserviert der Ausdruck `JFrame myFrame` den Namen `myFrame`. Auf derselben Codezeile erstellt der Ausdruck `new JFrame()` ein neues Objekt (eine Instanz der Klasse `JFrame`). Und zum Schluss sorgt das Gleichheitszeichen dafür, dass `myFrame` auf das neue Objekt verweist. (Man nennt so etwas auch »es referenziert das neue Objekt«.) Es kann sehr wichtig sein zu wissen, dass die beiden Wörter `new JFrame` ein neues Objekt erstellen. Mehr zu Objekten finden Sie in [Kapitel 7](#).

Ein Eintopf aus primitiven Typen

Da ich Frames erwähne, was ist das eigentlich? Ein *Frame* ist ein Fenster, das eine bestimmte Höhe und Breite hat und auf dem Bildschirm Ihres Computers eine bestimmte Position einnimmt. Deshalb können Sie, tief verborgen in der Deklaration der Klasse `JFrame`, eine Variablendeklaration auftun, die so ähnlich aussieht wie das hier:

```
int width;  
int height;  
int x;  
int y;
```

Ein weiteres Beispiel ist: `Time`. Eine Instanz der Klasse `Time` kann eine Stunde (eine Zahl von 1 bis 12), Minuten (von 0 bis 59) und einen Buchstaben (*a* für a. m. und *p* für p. m.) haben.

```
int hour;  
int minutes;  
char amOrPm;
```

Wie Sie sehen, ist dieses tolle Ding, das sich Java-API-Klasse nennt, nichts anderes als eine Sammlung von Deklarationen. Einige dieser Deklarationen sind Variablendeklarationen und einige dieser Variablendeklarationen verwenden primitive Typen, während andere Referenztypen nutzen. Diese Referenztypen wiederum stammen von anderen Klassen, deren Deklarationen ebenfalls Variablen enthalten. Und so weiter und so weiter. Aber letztendlich lässt sich alles auf primitive Typen reduzieren.

Eine Importdeklaration

In der Praxis hat es sich bewährt, als Erstes zu erklären, was man plant. Stellen Sie sich folgende Unterrichtsstunde vor:

*»In unserem Kurs zur Geschichte des Films wollen wir uns mit der Karriere des Schauspielers **Lionel Herbert Blythe Barrymore** beschäftigen.*

***Barrymore**, der in Philadelphia geboren wurde, wirkte in mehr als 200 Filmen mit, zu denen auch Ist das Leben nicht schön?, Key Largo und Dr. Kildare's Wedding Day gehören. Außerdem war **Barrymore** Schriftsteller, Komponist und Regisseur. Im Radio war er jedes Jahr die Stimme von Ebenezer Scrooge ...«*

Interessant, nicht wahr? Vergleichen Sie nun die Absätze oben mit einem Vortrag, bei dem der Redner nicht damit beginnt, sein Thema vorzustellen:

»Herzlich willkommen zur Geschichte des Films.

***Lionel Barrymore**, der in Philadelphia geboren wurde, wirkte in mehr als 200 Filmen mit, zu denen auch Ist das Leben nicht schön?, Key Largo und Dr. Kildare's Wedding Day gehören. Außerdem war **Barrymore (nicht Ethel, John oder Drew)** Schriftsteller, Komponist und Regisseur. Im Radio war **Lionel Barrymore** jedes Jahr die Stimme von Ebenezer Scrooge ...«*

Ohne eine saubere Einführung muss Sie ein Redner immer wieder daran erinnern, dass es um Lionel Barrymore und nicht um einen anderen Barrymore geht. Dasselbe gilt für ein Java-Programm. Schauen Sie sich noch einmal [Listing 4.6](#) an:

```
import javax.swing.JFrame;  
public class ShowAFrame {  
    public static void main(String args[]) {  
        JFrame myFrame = new JFrame();
```

In [Listing 4.6](#) kündigen Sie in der »Einführung des Programms« (in der *import-Deklaration*) an, dass Sie in Ihrer Java-Klasse JFrame verwenden wollen. Sie machen deutlich, was Sie wollen, indem Sie JFrame mit dem vollständigen Namen javax.swing.JFrame aufrufen. (Begann nicht auch der erste Redner mit dem vollständigen Namen *Lionel Herbert Blythe*

Barrymore?) Nachdem Sie in der `import`-Deklaration Ihre Absicht angekündigt haben, können Sie von da an im Code Ihrer Java-Klasse problemlos die Kurzform `JFrame` verwenden.

Wenn Sie auf eine `import`-Deklaration verzichten, müssen Sie den vollständigen Namen `javax.swing.JFrame` immer dann wiederholen, wenn Sie *JFrame* in Ihrem Code verwenden. Ohne eine `import`-Deklaration sähe der Code in [Listing 4.6](#) so aus:

```
public class ShowAFrame {  
    public static void main(String args[]) {  
        javax.swing.JFrame myFrame =  
            new javax.swing.JFrame();  
        String myTitle = "Leerer Frame";  
        myFrame.setTitle(myTitle);  
        myFrame.setSize(3200, 200);  
        myFrame.setDefaultCloseOperation  
            (javax.swing.JFrame.EXIT_ON_CLOSE);  
        myFrame.setVisible(true);  
    }  
}
```



Die Einzelheiten dieses Import-Gedöns können ziemlich haarig sein. Glücklicherweise haben viele IDEs ausgezeichnete Hilfe-Funktionen, die Fragen zu `import`-Deklarationen beantworten.

Importdeklarationen lassen sich nicht in einem Abschnitt dieses Buches erklären, weil sie zu vielschichtig sind. Sie finden in den [Kapiteln 5](#), [9](#) und [10](#) weitere Informationen zu diesem Deklarationstyp.

Mit Operatoren neue Werte erstellen

Wer kann einen besser beruhigen als Ihr alter Freund, das Pluszeichen? Es war das erste Element, das Sie in der Grundschule im Rechenunterricht kennengelernt haben. So gut wie jeder weiß, wie 2 und 2 addiert werden. Und in der deutschen

Sprache sagt »Der kann wohl 2 und 2 nicht zusammenzählen« ziemlich viel über das Gemüt eines Menschen aus. Und jedes Mal, wenn Sie ein Pluszeichen sehen, sagt eine Zelle in Ihrem Gehirn: »Dem Himmel sei Dank – es könnte viel schlimmer kommen.«

Auch Java kennt ein Pluszeichen. Sie können mit ihm verschiedene Dinge machen. So können Sie zum Beispiel das Pluszeichen verwenden, um zwei Zahlen zu addieren:

```
int apples, oranges, fruit;  
apples = 5;  
oranges = 16;  
fruit = apples + oranges;
```

Sie können das Pluszeichen aber auch verwenden, um `String`-Werte aneinanderzuhängen:

```
String startOfChapter =  
    "Es ist drei Uhr morgens. Ich träume vom " +  
    "Geschichtsunterricht, den ich in der Schule " +  
    "verpasst habe.";   
System.out.println(startOfChapter);
```

So etwas kann sich als sehr praktisch erweisen, weil es in Java nicht erlaubt ist, einen `String` zu definieren, der über mehrere Zeilen geht.

Das bedeutet, dass der folgende Code nicht funktionieren würde:

```
String startOfChapter =  
    "Es ist drei Uhr morgens. Ich träume vom  
    Geschichtsunterricht, den ich in der Schule  
    verpasst habe.";   
System.out.println(startOfChapter);
```



Sie hängen `String`-Werte natürlich nicht hintereinander, sondern Sie *verknüpfen* sie.

Sie können das Pluszeichen sogar dafür verwenden, um Zahlen neben `String`-Werte zu setzen:

```
int apples, oranges, fruit;  
apples = 5;  
oranges = 16;  
fruit = apples + oranges;
```

```
System.out.println("Sie haben " + fruit +  
                    " Stücke Obst.");
```

Natürlich gibt es auch das alte Minuszeichen (wenn auch nicht für String-Werte).

```
apples = fruit - oranges;
```

Verwenden Sie für die Multiplikation ein Sternchen (*) und für die Division einen Schrägstrich (/).

```
double rate, pay;  
int hours;  
rate = 6.25;  
hours = 35;  
pay = rate * hours;  
System.out.println(pay);
```

[Listing 4.3](#) bietet ein Beispiel für eine Division.



Wenn Sie einen `int`-Wert durch einen anderen `int`-Wert teilen, erhalten Sie wieder einen `int`-Wert. Dabei rundet der Computer nicht. Stattdessen schneidet er bei der Division entstehende Reste einfach ab. Wenn Sie in Ihrem Programm `System.out.println(11 / 4)` aufnehmen, gibt der Computer 2 und nicht 2.75 aus. Um dieses Problem zu umgehen, sollten Sie die Zahl, durch die Sie teilen (am besten beide Zahlen), als `double` deklarieren. Wenn Sie in Ihr Programm `System.out.println(11.0 / 4)` aufnehmen, wird als Ergebnis 2.75 ausgegeben.

Ein anderer nützlicher Operator ist der *Rest-Operator*. Sein Symbol ist das Prozentzeichen (%). Wenn Sie `System.out.println(11 % 4)` in Ihr Programm aufnehmen, gibt der Computer 3 aus. Warum? 4 geht wer-weiß-wie-offt in 11, und es bleibt ein Rest von 3. Es hat sich gezeigt, dass der Rest-Operator ziemlich nützlich sein kann, wie Sie [Listing 4.7](#) entnehmen können.

```
import static java.lang.System.out;  
public class MakeChange {  
    public static void main(String args[]) {  
        int total = 248;  
        int quarters = total / 25;
```

```

        int whatsLeft = total % 25;
        int dimes = whatsLeft / 10;
        whatsLeft = whatsLeft % 10;

        int nickels = whatsLeft / 5;
        whatsLeft = whatsLeft % 5;
        int cents = whatsLeft;
        out.println("Sie wechseln "total + " $-
Cents in");
        out.println(quarters + " Quarters");
        out.println(dimes + " Dimes");
        out.println(nickels + " Nickels");
        out.println(cents + " Cents");
    }
}

```

[Listing 4.7](#): Geld wechseln

[Abbildung 4.11](#) zeigt, was das Programm in [Listing 4.7](#) ausgibt. Sie beginnen mit 248 Cents. Dann dividiert

```
quarters = total / 25
```

248 durch 25, was 9 ergibt. Oder anders ausgedrückt: Sie können aus 248 Cents 9 Quarters machen. Nun dividiert

```
whatsLeft = total % 25
```

noch einmal 248 durch 25 und packt sich den Rest von 23 in whatsLeft. Jetzt können Sie mit dem nächsten Schritt weitermachen, bei dem es darum geht, so viele Dimes wie möglich aus 23 Cents herauszuholen.



[Abbildung 4.11](#): 2,48 Dollar werden gewechselt.

Der Code in [Listing 4.7](#) wechselt Geld, das in US-Währung vorliegt, mit der Stückelung 1 Cent, 5 Cent (ein Nickel), 10 Cent

(ein Dime) und 25 Cent (ein Quarter). Die Klasse `MakeChange` nimmt die ursprünglichen 248 Cent und gibt Ihnen die entsprechende Stückelung als Wechselgeld zurück. Dabei erhalten Sie *die kleinste Anzahl an Münzen*, die aufaddiert 248 Cent ergeben. Wenn Sie in den Code ein wenig Arbeit stecken, funktioniert er mit allen Münzen dieser Erde. Sie addieren dabei immer Münztypen bis zu einer vorgegebenen Summe. Es sind nun aber in einigen Ländern auch Stückelungen bei Münzen denkbar, die dafür sorgen, dass vom Programm nicht immer *die kleinste Anzahl an Münzen* zurückgegeben wird. Ich suche noch nach einem Beispiel, das `MakeChange` daran hindert, die beste Antwort zu geben. Sollten Sie in solch einem Land wohnen, senden Sie mir bitte eine E-Mail (JavaForDummies@allmycode.com).

Einmal initialisieren, mehrfach zuweisen

[Listing 4.7](#) hat drei Zeilen, die Werte in die Variable `whatsLeft` (deutsch *wasBleibtÜbrig*) packen.

```
int whatsLeft = total % 25;
whatsLeft = whatsLeft % 10;
whatsLeft = whatsLeft % 5;
```

Nur bei einer dieser Zeilen handelt es sich um eine Deklaration. Die anderen beiden Zeilen sind Zuweisungsbefehle. Das geht auch so in Ordnung, denn Sie können dieselbe Variable nur einmal deklarieren (zumindest nicht, ohne etwas zu erstellen, das sich *Block* nennt). Wenn Sie Murks machen und in [Listing 4.7](#)

```
int whatsLeft = total % 25;
int whatsLeft = whatsLeft % 10;
```

schreiben, erscheint bei dem Versuch, das Programm zu kompilieren, eine Fehlermeldung, die aussagt, dass `whatsLeft` bereits definiert ist. (Wenn Sie herausfinden wollen, was ein Block ist, siehe [Kapitel 5](#). In [Kapitel 10](#) gehe ich näher auf das erneute Deklarieren von Variablen ein.)

Import-Deklarationen: die hässliche Wahrheit

Beachten Sie die `import`-Deklaration zu Beginn von [Listing 4.7](#):

```
import static java.lang.System.out;
```

Vergleichen Sie dies mit der `import`-Deklaration zu Beginn von [Listing 4.6](#):

```
import javax.swing.JFrame;
```

Dadurch, dass ich [Listing 4.7](#) die Zeile

`import static java.lang.System.out;` hinzugefügt habe, schaffe ich es, dass der Rest des Codes etwas leichter zu lesen ist, und ich vermeide lange Java-Befehle, die über mehr als eine Zeile gehen. Sie müssen so nicht vorgehen. Wenn Sie die Zeile `import static java.lang.System.out;` entfernen und den Code großzügig mit `System.out.println` würzen, funktioniert er genauso gut.

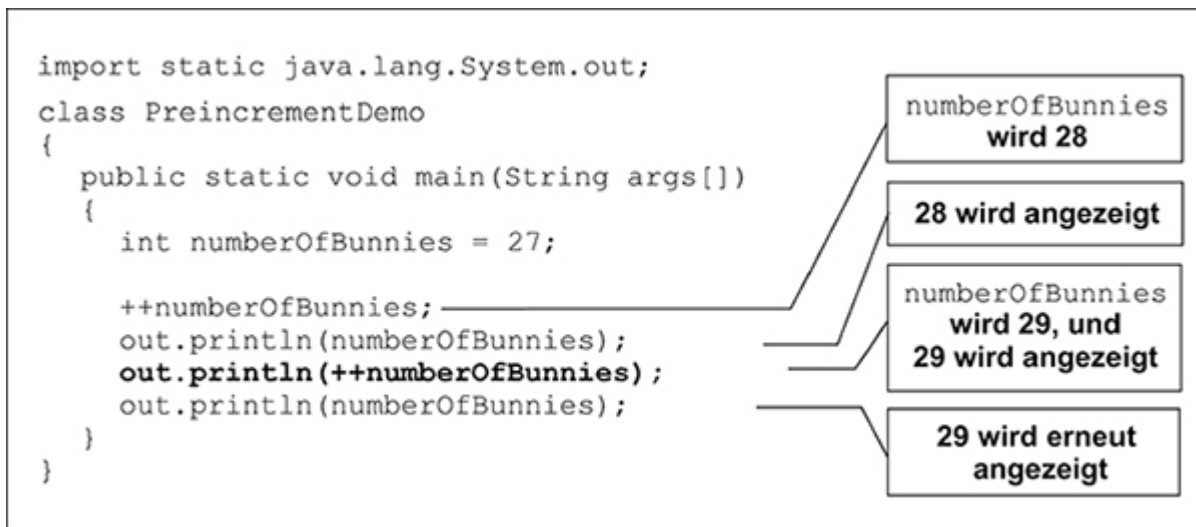
Eine Frage: Warum enthält eine Deklaration das Wort `static` und die andere nicht? Wenn ich ehrlich sein soll, wünschte ich mir, nicht gefragt zu haben.

Um das zu erfahren, was sich hinter `static` wirklich verbirgt, müssen Sie [Kapitel 10](#) zumindest teilweise lesen. Und wenn ich weiterhin ehrlich sein soll, empfehle ich Ihnen nicht, schon einmal zum Abschnitt über `static` zu springen, wenn Ihr Kreislauf schwach ist, Sie schwanger sind oder keine Erfahrung mit anderen Programmiersprachen haben. Im Moment sollte das Wissen ausreichen, dass sich [Kapitel 10](#) leicht lesen lässt, wenn Sie Ihre Tour durch Teil III dieses Buches hinter sich haben. Und wenn Sie dann entscheiden müssen, ob Sie in einer `import`-Deklaration das Wort `static` verwenden müssen, denken Sie an diese Tipps:

- ✓ Der weitaus größte Teil der `import`-Deklarationen in Java-Programmen benötigt das Wort `static` nicht.
- ✓ Ich verwende in diesem Buch `import static` ausschließlich in Verbindung mit `System.out`. (Nun gut, fast ausschließlich.)
- ✓ Die meisten `import`-Deklarationen verwenden `static` nicht, weil sie meistens Klassen importieren. Unglücklicherweise ist `System.out` nicht der Name einer Klasse.

Inkrement- und Dekrement-Operatoren

Java verfügt über einige nette kleine Operatoren, die das Leben (des Prozessors des Computers, Ihres Gehirns und Ihrer Fingerchen) wesentlich erleichtern. Insgesamt gibt es vier dieser Operatoren – zwei Inkrement- und zwei Dekrement-Operatoren. Die Inkrement-Operatoren addieren 1, während die Dekrement-Operatoren 1 abziehen. Die Inkrement-Operatoren verwenden doppelte Pluszeichen (++) und die Dekrement-Operatoren verwenden doppelte Minuszeichen (--). Um zu sehen, wie diese Operatoren arbeiten, folgen nun einige Beispiele. Das erste befindet sich in [Abbildung 4.12](#).



[Abbildung 4.12](#): Ein Präinkrement verwenden

[Abbildung 4.13](#) zeigt, was ausgegeben wird, wenn das Programm in [Abbildung 4.12](#) ausgeführt wird. Bei diesem wenig ereignisreichen Lauf wird die Anzahl der Bunnies (deutsch *Häschen*) dreimal angezeigt.

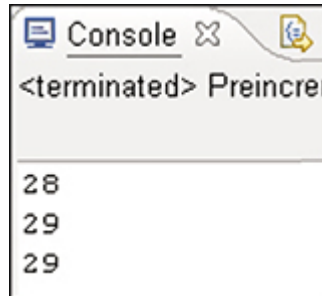


Abbildung 4.13: Der Code aus [Abbildung 4.12](#) wird ausgeführt.

Das doppelte Pluszeichen wird je nach seiner Position unterschiedlich bezeichnet. Wenn Sie es vor eine Variable setzen, wird es *Präinkrement*-Operator genannt. (*Prä* stammt vom Lateinischen *prae*, deutsch *vor*, ab.) Das Wort *vor* hat zwei Bedeutungen:

- ✓ Sie setzen ++ vor die Variable.
- ✓ Der Computer fügt der Variablen 1 hinzu, bevor diese irgendwo im Befehl verwendet wird.

Um dies zu verstehen, schauen Sie sich einmal die fett gedruckte Zeile in [Abbildung 4.12](#) an. Der Computer fügt `numberOfBunnies` 1 hinzu (und erhöht dabei den Wert von `numberOfBunnies` auf 29), bevor er 29 auf dem Bildschirm ausgibt.



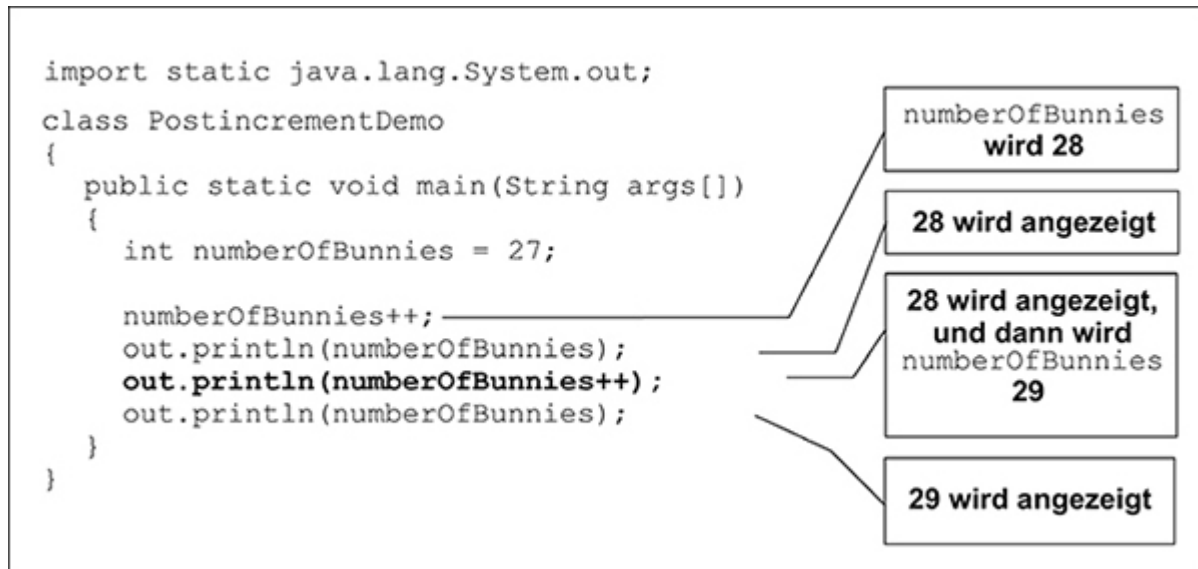
Bei `out.println(++numberOfBunnies)` fügt der Computer `numberOfBunnies` 1 hinzu, bevor der neue Wert von `numberOfBunnies` auf dem Bildschirm ausgegeben wird.

Eine Alternative zum Präinkrement ist das *Postinkrement*. (*Post* steht für *hinter*, *nach*.) Auch dieses Wort hat zwei Bedeutungen:

- ✓ Sie setzen ++ hinter die Variable.
- ✓ Der Computer erhöht den Wert der Variablen um 1, nachdem diese irgendwo im Befehl verwendet worden ist.

Damit dieses Verhalten deutlicher wird, sollten Sie einen Blick auf die in [Abbildung 4.14](#) fett gedruckte Zeile werfen. Der Computer

gibt auf dem Bildschirm den alten Wert von `numberOfBunnies` aus (der 28 beträgt) und addiert danach 1 zu `numberOfBunnies`, das dadurch zu 29 wird.



[Abbildung 4.14](#): Ein Postinkrement verwenden



Bei `out.println(numberOfBunnies++)` fügt der Computer `numberOfBunnies` 1 hinzu, nachdem der alte Wert von `numberOfBunnies` auf dem Bildschirm ausgegeben worden ist.

[Abbildung 4.15](#) zeigt einen Lauf des Programms in [Abbildung 4.14](#). Vergleichen Sie diese Abbildung mit [Abbildung 4.13](#):

- ✓ Bei dem Präinkrement in [Abbildung 4.13](#) lautet die zweite Zahl 29.
- ✓ Bei dem Postinkrement in [Abbildung 4.15](#) lautet die zweite Zahl 28.

In [Abbildung 4.15](#) wird 29 erst am Ende des Programmlaufs auf dem Bildschirm angezeigt, wenn der Computer ein letztes Mal `out.println(numberOfBunnies)` ausführt.

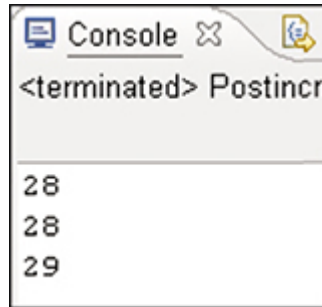


Abbildung 4.15: Der Code in [Abbildung 4.14](#) wird ausgeführt.



Stehen Sie auch vor der Frage, wann ein Post- und wann ein Präinkrement verwendet werden soll? Hören Sie auf zu grübeln. Die meisten Programmierer verwenden Postinkremente. Sie stoßen in einem typischen Java-Programm öfter auf Konstrukte wie `numberOfBunnies++` als auf `++numberOfBunnies`.

Befehle und Ausdrücke

Sie können die Operatoren Prä- und Postinkrement und Prä- und Postdekrement auf zwei Arten beschreiben: so, dass jedermann diese Begriffe versteht, oder richtig. Ich habe mich in diesem Abschnitt dafür entschieden, (mithilfe der Begriffe *vor* und *nach* oder *hinter*) den Weg zu gehen, der allgemein verständlich sein sollte. Unglücklicherweise ist dies nicht der wirklich richtige Weg. Wenn Sie `++` oder `--` sehen, können Sie an einen Zeitablauf denken. Leider verwenden Programmierer `++` oder `--` gelegentlich auf so eine verdrehte Weise, dass *vor* und *nach* jeden Sinn verlieren. Deshalb sollten Sie bei Problemen mit diesen Operatoren in Begriffen wie Befehle und Ausdrücke denken.

Zunächst sollten Sie daran denken, dass ein Befehl den Computer anweist, etwas zu tun, und dass ein Ausdruck einen Wert hat. (Ich behandle Befehle in [Kapitel 3](#) und Ausdrücke irgendwo in diesem Kapitel.) In welche Kategorie gehört nun `numberOfBunnies`? Die überraschende Antwort ist: in beide. Der Java-Code `numberOfBunnies++` ist sowohl ein Befehl als auch ein Ausdruck.

Nehmen wir einmal an, dass der Wert von `numberOfBunnies` 28 beträgt, bevor der Computer den Code `out.println(numberOfBunnies++)` ausführt.

✓ `numberOfBunnies++` weist als Befehl den Computer an, `numberOfBunnies` um 1 zu erhöhen.

✓ Als Ausdruck beträgt der Wert von `numberOfBunnies` 28 und nicht 29.

Obwohl der Computer den Wert von `numberOfBunnies` um 1 erhöht hat, bedeutet der Code `out.println(numberOfBunnies++)` tatsächlich `out.println(28)`.

Nun gilt alles, was Sie gerade über `numberOfBunnies++` gelesen haben, auch für `++numberOfBunnies`. Der einzige Unterschied ist, dass sich ein Ausdruck wie `++numberOfBunnies` intuitiver begreifen lässt.

✓ `++numberOfBunnies` weist als Befehl den Computer an, `numberOfBunnies` um 1 zu erhöhen.

✓ Als Ausdruck beträgt der Wert von `++numberOfBunnies` 29.

Bei dem Befehl `out.println(++numberOfBunnies)` erhöht der Computer den Wert der Variablen `numberOfBunnies` um 1, und der Code `out.println(++numberOfBunnies)` bedeutet in Wirklichkeit `out.println(29)`.

Java verfügt zusätzlich zu Prä- und Postinkrement über zwei Operatoren, die `--` verwenden. Diese beiden Operatoren heißen *Prädecrement* und *Postdecrement*.

✓ Bei einem Prädecrement (`--numberOfBunnies`) zieht der Computer 1 vom Wert der Variablen ab, bevor die Variable im Befehl verwendet wird.

✓ Bei einem Postdecrement (`numberOfBunnies--`) zieht der Computer 1 vom Wert der Variablen ab, nachdem die Variable im Befehl verwendet worden ist.



Sie können denselben Effekt wie mit `++numberOfBunnies` erzielen, wenn Sie `numberOfBunnies = numberOfBunnies + 1` schreiben. Daraus schließen einige Leute, dass die Operatoren `++` und `--` nur Tastenanschläge einsparen sollen – damit sich die

armen Fingerchen nicht überarbeiten. Dies ist vollkommen falsch. Der beste Grund, um ++ zu verwenden, ist, die ineffiziente und fehlerbehaftete Doppeleingabe von Variablennamen wie `numberOfBunnies` in demselben Befehl zu vermeiden. Wenn Sie nur einmal `numberOfBunnies` schreiben müssen (wie das bei der Verwendung von ++ oder -- der Fall ist), muss der Computer nur einmal herausfinden, was `numberOfBunnies` bedeutet. Außerdem besteht dann nur einmal die Gefahr, den Namen der Variablen falsch einzugeben. Bei einfachen Ausdrücken wie `numberOfBunnies++` macht sich dieser Vorteil kaum bemerkbar. Aber wenn Sie kompliziertere Ausdrücke wie `inventarAnzahl[(mengeErhalten-- *anzahlProKiste+17)]++` schreiben, bedeutet die Verwendung von ++ oder -- einen erheblichen Gewinn an Effizienz und Genauigkeit.

Zuweisungsoperatoren

Falls Sie den vorherigen Abschnitt gelesen haben, der von Operatoren handelt, die 1 hinzufügen, haben Sie sich vielleicht gefragt, ob Sie diese Operatoren nicht so ändern können, dass sie nicht nur 1, sondern 2, 5 oder 10000000 hinzufügen. Besteht die Möglichkeit, als Java-Programmierer `numberOfBunnies++++` zu schreiben? Vergessen Sie's. Wenn Sie das ausprobieren, erscheint bei dem Versuch, den Code zu kompilieren, eine Fehlermeldung.

Was also können Sie tun? Glücklicherweise verfügt Java über viele Zuweisungsoperatoren, die nur auf Sie warten. Sie können rechentechisch mit einem *Zuweisungsoperator* alles erledigen, was Sie wollen: addieren, subtrahieren, multiplizieren oder dividieren. Und Sie haben die Möglichkeit, mit diesen Operatoren weitere coole Operationen auszuführen. [Listing 4.8](#) enthält eine bunte Mischung von Zuweisungsoperatoren (den Dingen mit dem Gleichheitszeichen). [Abbildung 4.16](#) zeigt, was dieses Programm auf dem Bildschirm ausgibt.

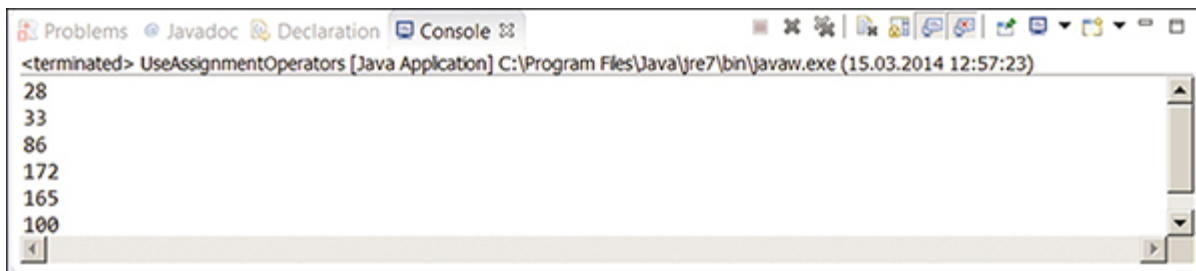
```
public class UseAssignmentOperators {
```

```

    public static void main(String args[]) {
        int numberOfBunnies = 27;
        int numberExtra = 53;
        numberOfBunnies += 1;
        System.out.println(numberOfBunnies);
        numberOfBunnies += 5;
        System.out.println(numberOfBunnies);
        numberOfBunnies += numberExtra;
        System.out.println(numberOfBunnies);
        numberOfBunnies *= 2;
        System.out.println(numberOfBunnies);
        System.out.println(numberOfBunnies -= 7);
        System.out.println(numberOfBunnies = 100);
    }
}

```

[Listing 4.8](#): Zuweisungsoperatoren



[Abbildung 4.16](#): Der Code in [Listing 4.8](#) wird ausgeführt.

[Listing 4.8](#) zeigt, wie vielseitig Javas Zuweisungsoperatoren sind. Sie können mit Zuweisungsoperatoren beliebige Zahlen zu einer Variablen addieren, von ihr subtrahieren, mit ihr multiplizieren oder durch sie dividieren. Beachten Sie, wie `+=5` 5 zu `numberOfBunnies` hinzufügt, und wie `*=2` `numberOfBunnies` mit 2 multipliziert. Sie sind sogar in der Lage, den Wert eines anderen Ausdrucks (in [Listing 4.8](#) `numberExtra`) zu nehmen, um mit ihm eine Rechenoperation durchzuführen.

Die letzten beiden Zeilen in [Listing 4.8](#) veranschaulichen eine besondere Funktion von Javas Zuweisungsoperatoren. Sie können einen Zuweisungsoperator als Teil eines größeren Java-Befehls verwenden. In der vorletzten Zeile in [Listing 4.8](#)

subtrahiert der Operator `7` von `numberOfBunnies` und verringert damit den Wert von `numberOfBunnies` von `172` auf `165`. Dann wird der ganze Zuweisungskram in einen Aufruf von `System.out.println` gepackt, was dazu führt, dass auf dem Bildschirm `165` ausgegeben wird.

Die letzte Zeile von [Listing 4.8](#) zeigt nun noch, dass Sie Dinge dieser Art auch mit dem guten alten Java-Gleichheitszeichen veranstalten können. Der Ausdruck, den ich am Anfang dieses Kapitels als Zuweisungsbefehl beschreibe, ist in Wirklichkeit einer der Zuweisungsoperatoren, die ich in diesem Kapitel erkläre. Daraus folgt, dass Zuweisungen, die Sie vornehmen, problemlos auch Teil eines umfangreicheren Befehls sein können.



Jede Verwendung eines Zuweisungsoperators erfüllt zwei Aufgaben – eine als Befehl und eine als Ausdruck. In beiden Fällen ist der Wert des Ausdrucks gleich dem Wert, den Sie zuweisen. So hat zum Beispiel `numberOfBunnies` vor der Ausführung von

```
System.out.println(numberOfBunnies -= 7)
```

den Wert `172`. Als Befehl weist `numberOfBunnies -= 7` den Computer an, von `numberOfBunnies` `7` abzuziehen (wodurch sich der Wert von `172` auf `165` verringert). Als Ausdruck hat `numberOfBunnies -= 7` den Wert `165`.

```
System.out.println(numberOfBunnies -= 7)
```

bedeutet tatsächlich `System.out.println(165)`.

Eine ausführlichere Erklärung dieser Dinge finden Sie weiter vorn in diesem Kapitel im Kasten *Befehle und Ausdrücke*.

5

Den Programmablauf mit entscheidungsfindenden Befehlen steuern

In diesem Kapitel

- ▶ Befehle schreiben, die zwischen Alternativen unterscheiden
 - ▶ Befehle verschachteln
 - ▶ Unter vielen Alternativen auswählen
-

Die Fernsehserie *Dennis the Menace* wurde von 1959 bis 1963 in den USA von CBS (und in den 1980er Jahren in Deutschland unter dem Titel *Dennis*) ausgestrahlt. Ich erinnere mich an eine Folge, in der Mr. Wilson Schwierigkeiten hatte, eine wichtige Entscheidung zu fällen. Ich glaube, es ging dabei um so etwas wie einen Wechsel der Arbeitsstelle oder einen Umzug in eine andere Stadt. Auf jeden Fall sehe ich noch die Szene vor mir, wie Mr. Wilson in seinem Garten sitzt, an einem Glas Limonade nippt und den ganzen Nachmittag lang ins Nirgendwo starrt. Natürlich tauchte alle naslang der nervige Dennis auf, um Mr. Wilsons Ruhe und seinen Frieden zu stören – was für die Komik in dieser Situation sorgte.

Was mich an dieser Folge so beeindruckt hat (und was der Grund dafür ist, dass ich mich bis heute so genau daran erinnern kann), war die Beharrlichkeit, mit der Mr. Wilson versuchte, eine Entscheidung zu fällen. Dieser Kerl hat nicht weitergemacht, als ob nichts wäre, während in seinem Kopf die Gedanken über die Entscheidung verrückt spielten. Er saß ruhig in seinem Garten und schrieb sich in Gedanken die Vor- und die Nachteile auf. Wer fällt heute noch Entscheidungen auf diese Weise?

Damals war ich noch ziemlich jung. Ich war noch nicht gezwungen, eine tiefgreifende Entscheidung zu fällen, die meine Familie und mich betraf. Ich fragte mich, wie solch ein Entscheidungsfindungsprozess wohl sein würde. Würde es helfen, stundenlang stumm herumzusitzen? Würde ich meine Entscheidung sorgfältig abwägen und alle Optionen prüfen? Oder würde ich einfach einen Schuss ins Blaue abgeben, alles riskieren und impulsiv handeln? Damals hatte ich darauf keine Antwort.

Entscheidungen fällen (»if«-Befehle in Java)

Wenn Sie Computerprogramme schreiben, müssen Sie sich ständig für die eine oder die andere Möglichkeit entscheiden. Hat der Benutzer das Kennwort richtig eingegeben? Wenn ja, darf er arbeiten, anderenfalls wirf ihn aus dem System. Java benötigt deshalb eine Möglichkeit, um im Programm Weggabelungen bereitzustellen. Glücklicherweise gibt es so etwas. Es nennt sich `if`-Befehl.

Eine Zahl raten

[Listing 5.1](#) demonstriert den Einsatz eines `if`-Befehls. [Abbildung 5.1](#) zeigt, was das Programm in [Listing 5.1](#) nach zweimaliger Ausführung ausgegeben hat.

```
import static java.lang.System.out;
import java.util.Scanner;
import java.util.Random;
public class GuessingGame {
    public static void main(String args[]) {
        Scanner keyboard = new Scanner(System.in);
        out.print("Geben Sie eine ganze Zahl "
            + "zwischen 1 und 10 ein: ");
        int inputNumber = keyboard.nextInt();
        int randomNumber = new Random().nextInt(10) +
1;

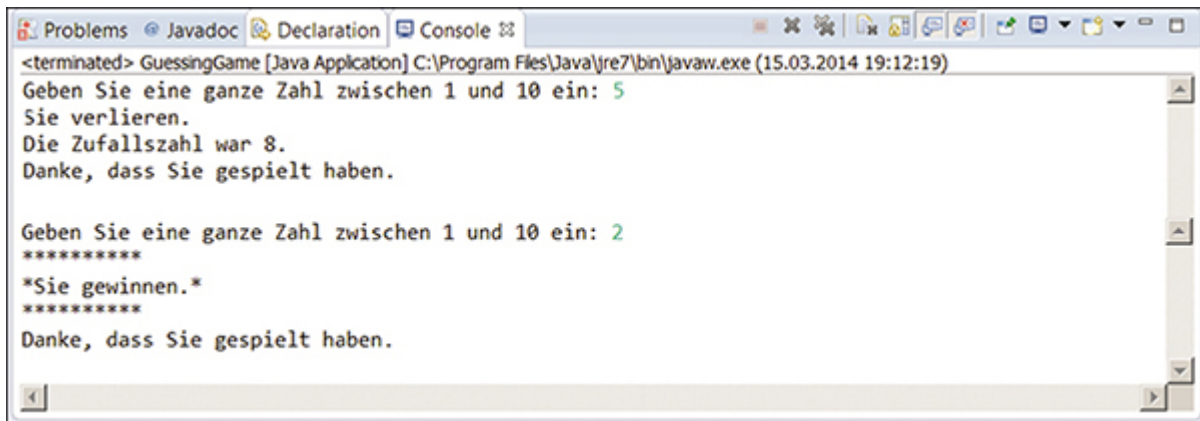
        if (inputNumber == randomNumber) {
            out.println("*****");
        }
    }
}
```

```

        out.println("*Sie gewinnen.*");
        out.println("*****");
    } else {
        out.println("Sie verlieren.");
        out.print("Die Zufallszahl war ");
        out.println(randomNumber + ".");
    }
    out.println("Danke, dass Sie gespielt haben."
);
    keyboard.close();
}
}

```

[Listing 5.1](#): Ein Ratespiel



[Abbildung 5.1](#): Zwei Ausführungen des Ratespiels

Das Programm in [Listing 5.1](#) spielt mit dem Benutzer ein Ratespiel. Das Programm erhält vom Benutzer eine (geratene) Zahl und generiert eine Zufallszahl zwischen 1 und 10. Wenn die Zahl, die der Benutzer eingegeben hat, mit der Zufallszahl übereinstimmt, gewinnt der Benutzer. Anderenfalls verliert er, und das Programm zeigt dem Benutzer, wie die Zufallszahl aussieht.

Tastatureingaben kontrollieren

Zusammengefasst sieht es so aus, dass in [Listing 5.1](#) die Zeilen

```

import java.util.Scanner;
Scanner keyboard = new Scanner(System.in);
int inputNumber = keyboard.nextInt();

```


die Zahl übernehmen, die ein Benutzer auf der Tastatur des Computers eingibt. Die letzte der drei Zeilen packt diese Zahl in eine Variable mit dem Namen `inputNumber`. Lassen Sie sich nicht dadurch beirren, dass diese Zeilen ein wenig kompliziert aussehen. Sie können sie immer dann Wort für Wort übernehmen, wenn Sie Tastatureingaben auslesen wollen. Nehmen Sie die ersten beiden Zeilen (die `import`- und `Scanner`-Zeilen) nur einmal in Ihr Programm auf. Und immer, wenn der Benutzer im Programm eine Zahl eingeben soll, fügen Sie eine Zeile mit `nextInt` hinzu (siehe die letzte der drei vorstehenden Codezeilen).

Von all den Namen in diesen drei Zeilen gehen nur zwei auf mein Konto: `inputNumber` und `keyboard`. Alle anderen Namen sind Teil von Java. Wenn ich kreativ sein möchte, kann ich die Zeilen auch so schreiben:

```
import java.util.Scanner;
    Scanner readingThingie = new Scanner(System.in);

    int valueTypedIn = readingThingie.nextInt();
```

Ich könnte die `import`-Deklaration meines Programms auch so aufpolieren, wie ich das in den [Listings 5.2](#) und [5.3](#) mache, aber mehr kann ich eigentlich nicht tun.

Wenn Sie weiterlesen, werden Sie anfangen, die Muster hinter diesen drei Codezeilen zu erkennen. Ich möchte Sie hier nicht mit Informationen erschlagen. Im Moment sollte es ausreichen zu wissen, dass Sie diese drei Codezeilen kopieren können, und sich Folgendes zu merken:

✓ **Wenn Sie `java.util.Scanner` importieren, verwenden Sie das Wort `static` nicht.**

Der Import von `Scanner` unterscheidet sich vom Import von `System.out`. Wenn Sie `java.lang.System.out` importieren, verwenden Sie das Wort `static` (siehe [Listing 5.1](#)). Der Unterschied liegt im Code, denn `Scanner` ist der Name einer Klasse, während dies bei `System.out` nicht der Fall ist. In einem der Kästen in [Kapitel 4](#) beschreibe ich in aller

Kürze, wie `static` verwendet wird, und in [Kapitel 10](#) gehe ich tiefer auf dieses Thema ein.

✓ **Der Name `System.in` steht für die Tastatur.**

Um Zeichen von einer anderen Quelle als der Tastatur zu erhalten, schreiben Sie in die Klammern etwas anderes als `System.in`. Was das sein kann, beschreibe ich in [Kapitel 8](#).



Ich habe in [Listing 5.1](#) die Entscheidung gefällt, einer meiner Variablen den Namen `keyboard` zu geben. Dieser Name soll Sie, den Leser, daran erinnern, dass sich die Variable auf einen Haufen von Plastiktasten vor Ihnen bezieht. Die Namensgebung `keyboard` sagt Java nichts über Tasten aus Plastik oder Benutzereingaben. Andererseits erinnert der Name `System.in` Java immer an eben diese Plastikteile. In [Listing 5.1](#) verbindet der Code `Scanner keyboard = new Scanner(System.in)` den Namen `keyboard` mit den Plastiktasten, die wir kennen und lieben.

✓ **Wenn Sie erwarten, dass der Benutzer einen `int`-Wert (eine beliebige Ganzzahl) eingibt, verwenden Sie `nextInt()`.**

Wenn Sie davon ausgehen, dass der Benutzer einen `double`-Wert (eine Zahl mit Nachkommastellen) eingibt, verwenden Sie `nextDouble()`. Wenn Sie erwarten, dass der Benutzer `true` oder `false` eingibt, verwenden Sie `nextBoolean()`. Und wenn Sie erwarten, dass der Benutzer Wörter wie `Barry`, `Java` oder `Hallo` eingibt, verwenden Sie `next()`. Ein Beispiel für die Eingabe eines Wortes enthält [Listing 5.3](#), während [Listing 6.4](#) in [Kapitel 6](#) ein Beispiel für die Eingabe eines einzelnen Buchstabens enthält. Und in [Kapitel 8](#) finden Sie ein Beispiel dafür, wie ein Programm einen ganzen Satz (in einem Rutsch) einliest.



Dezimalzeichen unterscheiden sich von Land zu Land. In Deutschland stellt 10,5 (mit einem Komma) zehneinhalb dar, aber in den Vereinigten Staaten wird der gleiche Wert mit

einem Punkt als Dezimalzeichen geschrieben (10.5). In der persischen Sprache sieht ein Dezimalzeichen wie ein Bindestrich aus (der etwas tiefer sitzt als die Ziffern). Das Betriebssystem Ihres Computers speichert Informationen über das Land, in dem Sie leben, und Java liest diese Informationen aus, um zu entscheiden, wie *zehneinhalb* auszusehen hat, wenn Sie diesen Wert über die Tastatur eingeben. Wenn Sie ein Programm ausführen, in dem die Methode `nextDouble()` aufgerufen wird und Java mit einer `InputMismatchException` antwortet, überprüfen Sie Ihre Eingabe. Es könnte sein, dass Sie `10.5` eingegeben haben, während Ihr Ländercode `10,5` (oder eine andere Darstellung des Dezimalzeichens) verlangt.

✓ **Sie können von der Tastatur nacheinander verschiedene Werte erhalten.**

Verwenden Sie zu diesem Zweck mehrfach `keyboard.nextInt()`. Wie ein Programm mehr als einen Wert von der Tastatur empfängt, zeigt [Listing 5.4](#).

✓ **Immer wenn Sie Javas `Scanner` verwenden, sollten Sie nach dem letzten Aufruf von `nextInt` (oder `nextDouble` oder `nextWasAuchImmer`) die Methode `close()` aufrufen.**

In [Listing 5.1](#) lautet der letzte Befehl der Methode `main`

```
keyboard.close();
```

Dieser Befehl übernimmt das Organisatorische und trennt das Java-Programm von der Tastatur des Computers. Falls ich diesen Befehl in [Listing 5.1](#) vergessen hätte, wäre nichts Schreckliches passiert. Javas virtuelle Maschine (die JVM) ist normalerweise so freundlich, selbst hinter sich aufzuräumen. Es hat sich aber bewährt, mit dem Aufrufen von `close()` die Trennung von der Tastatur bewusst vorzunehmen. Es gibt sogar IDEs, die Warnungen anzeigen, wenn Sie die Anweisung `keyboard.close()` vergessen. Deshalb denke ich in den Beispielen dieses Buches immer daran, die Variable `Scanner` mit `close()` zu schließen (wobei ich Ihnen in

[Kapitel 12](#) einen zuverlässigeren Weg zeige, um `keyboard.close()` in ein Programm einzubinden).



Wenn Ihr Programm `System.out.println` aufruft, verwendet es den Bildschirm Ihres Computers. Warum rufen Sie eine `close()`-Methode dann nicht im Anschluss an `System.out.println` auf? Die Antwort ist nicht so einfach. In [Listing 5.1](#) verbindet sich Ihr eigener Code mit der Tastatur, indem `new Scanner (in)` aufgerufen wird. Später im Programm räumt der Code hinter sich selbst auf, indem er die Methode `close()` aufruft. Nun ist es so, dass Ihr Code mit `System.out.println` keine Verbindung zum Bildschirm herstellt. (Die Variable `out` verweist auf einen `PrintStream`, aber Sie rufen `new PrintStream()` als Vorbereitung für den Aufruf von `System.out.println` nicht auf.) Stattdessen verbindet sich die Java Virtual Machine für Sie mit dem Bildschirm. Der Code der Java Virtual Machine (um den Sie sich niemals kümmern müssen) enthält als Vorbereitung für Ihren Aufruf von `System.out.println` einen Aufruf von `PrintStream()`. Und da nun der Java-Code ein braves Stückchen Code ist, ruft die Java Virtual Machine schließlich `out.close()` ohne Ihr Zutun auf.

Zufallszahlen

Es ist überraschend schwer, einen echten Zufall zu erhalten. Der Mathematiker Persi Diaconis behauptete, dass eine Münze, die Sie mehrmals hintereinander in die Luft werfen und die dabei mit der Vorderseite nach oben zeigt, häufiger auf ihrer Rückseite als auf ihrer Vorderseite landet. Und wenn Sie die Münze mit ihrer Rückseite nach oben werfen, landet sie öfter auf ihrer Vorderseite als auf ihrer Rückseite. Noch nicht einmal Münzen sind fair.

Computer sind nicht besser als Münzen. Der Computer imitiert zufällige Folgen, denn letztendlich macht er nur das, was ihm gesagt wird, und das macht er unverfälscht und ohne sich aufhalten zu lassen. Wenn der Computer in [Listing 5.1](#)

```
import java.util.Random;
    int randomNumber = new Random().nextInt(10) +
    1;
```

ausführt, sieht es so aus, als ob er uns eine zufällig ausgewählte Zahl – eine Ganzzahl zwischen 1 und 10 – zurückgäbe. Das ist aber eine Täuschung. Der Computer folgt Anweisungen. Das hat nicht wirklich etwas mit Zufall zu tun, ist aber das Beste, was zu erreichen ist.

Ich kann Ihnen nur noch einmal den Tipp geben, diesen Code hinzunehmen. Machen Sie sich erst dann Gedanken über die Bedeutung von `new Random().nextInt`, wenn Sie mehr Erfahrung mit Java gesammelt haben. Kopieren Sie diesen Code einfach in Ihre eigenen Programme – und viel Vergnügen damit. Wenn die Zahlen von 1 bis 10 nicht zu Ihren Plänen passen, sollten Sie ruhig bleiben. Würfeln Sie ein wenig, indem Sie sich mit der Anweisung

```
int rollEmBaby = new Random().nextInt(6) + 1;
```

einen imaginären Würfel basteln. Wenn Sie diesen Befehl ausführen, liefert die Variable `rollEmBaby` Werte zwischen 1 und 6.

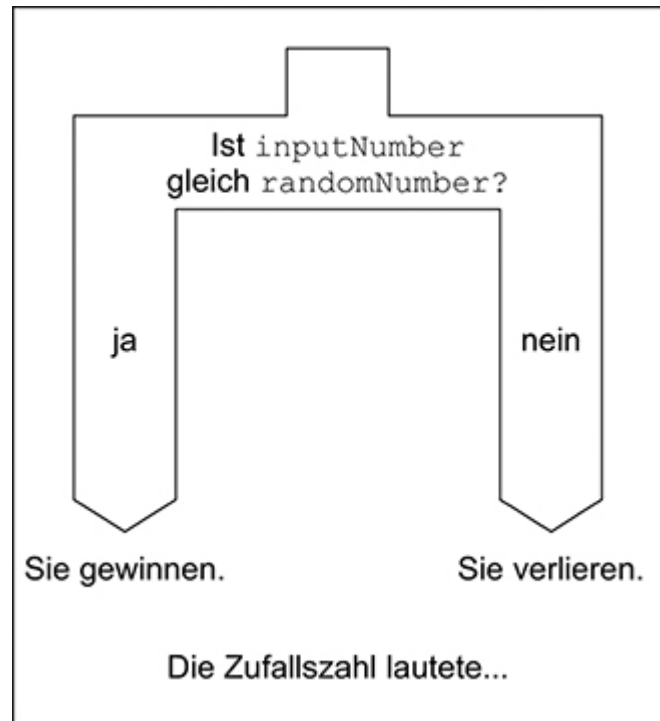
Der Befehl »if«

Im Zentrum von [Listing 5.1](#) steht ein Java-`if`-Befehl. Sie können diesen Befehl mit einer Weggabelung vergleichen (siehe [Abbildung 5.2](#)). Der Computer folgt einer von zwei Möglichkeiten – der, die Sie gewinnen ausgibt, oder der mit Sie verlieren. Der Computer entscheidet, welche Richtung Sie einschlagen, indem er eine *Bedingung* auf richtig oder falsch hin überprüft. In [Listing 5.1](#) lautet die Bedingung, die getestet werden soll,

```
inputNumber == randomNumber
```

Entspricht der Wert der eingegebenen Zahl `inputNumber` dem Wert der Zufallszahl `randomNumber`? Wenn die Bedingung wahr ist, wird das ausgeführt, was zwischen der Bedingung und dem Wort `else` steht. Stellt es sich aber heraus, dass die Bedingung falsch ist, führt der Computer die Anweisungen aus, die hinter

dem Wort `else` stehen. Auf jeden Fall führt der Computer danach den letzten `println`-Aufruf aus, der Danke, dass Sie gespielt haben ausgibt.



[Abbildung 5.2](#): Ein `if`-Befehl verzweigt sich wie eine Weggabelung.



Die Bedingung muss in einem `if`-Befehl in Klammern stehen. Natürlich ist eine Zeile wie `if (inputNumber == randomNumber)` kein vollständiger Befehl (so wie »Wenn ich keine Steuern zahle« kein vollständiger Satz ist). Aus diesem Grund darf die Zeile `if (inputNumber == randomNumber)` auch nicht mit einem Semikolon enden.



Manchmal, wenn ich über eine Bedingung schreibe, die gerade getestet wird, rutscht mir das Wort *Ausdruck* raus, obwohl *Bedingung* passender wäre. Aber das geht schon in Ordnung, weil jede Bedingung ein Ausdruck ist. Ein Ausdruck ist eben etwas, das einen Wert hat, und zweifellos hat eine Bedingung einen Wert. Der Wert der Bedingung ist entweder

`true` oder `false`. (Wenn Sie mehr über Ausdrücke und Bedingungen wie `true` und `false` wissen wollen, siehe [Kapitel 4](#).)

Das doppelte Gleichheitszeichen

Der `if`-Befehl in [Listing 5.1](#) enthält in seiner Bedingung ein doppeltes Gleichheitszeichen (`==`). Es ist nicht dasselbe, zwei Werte miteinander zu vergleichen, um herauszufinden, ob es sich dabei um dieselben Werte handelt, oder etwas mit etwas anderem gleichzusetzen. Deshalb unterscheidet sich auch das Symbol, das bei Gleichheit verwendet wird, von dem, das Sie in einer Zuweisung benutzen. Sie dürfen in der Bedingung eines `if`-Befehls das doppelte Gleichheitszeichen niemals durch ein einfaches Gleichheitszeichen ersetzen. Wenn Sie so etwas tun, funktioniert Ihr Programm nicht mehr. (Sie erhalten dann bei dem Versuch, den Code zu kompilieren, eine Fehlermeldung.)

Wenn Sie andererseits aber niemals den Fehler machen, in einer Bedingung ein einfaches Gleichheitszeichen zu verwenden, sind Sie nicht normal. Vor noch nicht allzu langer Zeit habe ich während eines Einführungskurses in Java geschworen, dass ich meinen Laserpointer verspeise, wenn niemand im Verlauf des Kurses den Fehler mit dem einfachen Gleichheitszeichen begehen würde. Dieses Versprechen konnte ich gefahrlos geben. Ich wusste, dass ich es niemals würde einlösen müssen. Und selbst wenn ich die ersten zehn Fehlermeldungen wegen des falschen Setzens einfacher Gleichheitszeichen aus meiner Wette herausgenommen hätte, wäre ich auch heute noch Laserpointerfrei. Jeder Programmierer begeht während seiner Karriere diesen Fehler mehrmals. Der Trick besteht nun nicht darin, den Fehler zu vermeiden, sondern ihn immer zu erkennen, wenn er passiert.

Nichts als Klammern

Der Befehl `if` in [Listing 5.1](#) besteht aus zwei Hälften – einer oberen und einer unteren Hälfte. Beide Teile haben bei mir Namen. Ich nenne sie den *if-Teil* (die obere Hälfte) und den *else-Teil* (die untere Hälfte).

Der `if`-Teil in [Listing 5.1](#) scheint mehr als einen Befehl zu enthalten. Ich erreiche dies dadurch, dass ich die drei Befehle des `if`-Teils von einem Paar geschweiffter Klammern einschließen lasse. Auf diese Weise bilde ich einen *Block*. Bei einem Block handelt es sich um eine Reihe von Anweisungen, die durch ein Paar geschweiffter Klammern zusammengehalten werden.

Mit diesem Block können drei Aufrufe von `println` sicher im `if`-Teil verstaubt werden. Durch die geschweiften Klammern werden die Zeilen mit den Sternchen und die Wörter `Sie gewinnen` nur dann angezeigt, wenn der Benutzer die Zahl richtig erraten hat.

Die Sache mit den Blöcken und den geschweiften Klammern gilt natürlich auch für den `else`-Teil. Immer wenn `inputNumber` nicht gleich `randomNumber` ist, führt der Computer im `else`-Teil drei Aufrufe von `print/println` aus. Um den Computer nun davon zu überzeugen, dass sich alle drei Aufrufe in der `else`-Klausel befinden, habe ich sie in einen Block gepackt. Oder anders ausgedrückt, ich umschließe die drei Aufrufe mit einem Paar geschweiffter Klammern.



Wenn man es genau nimmt, enthält [Listing 5.1](#) zwischen `if` und `else` nur einen Befehl und einen weiteren Befehl hinter dem `else`-Befehl. Der Trick besteht hier darin, dass Sie einen Block erhalten, wenn Sie eine Reihe von Befehlen innerhalb von geschweiften Klammern unterbringen. Und ein Block verhält sich nun einmal wie ein einziger Befehl. Das geht sogar so weit, dass Blöcke in der offiziellen Java-Dokumentation als Befehle abgehandelt werden. Deshalb ist der Block, der `Sie gewinnen` und Sternchen ausgibt, ein einzelner Befehl, der drei kleinere Befehle enthält.

Den Code bei »if«-Befehlen einrücken

In [Listing 5.1](#) sind die Aufrufe von `print` und `println` im `if`-Befehl eingerückt. (Dies betrifft die beiden Anweisungen `Sie`

gewinnen und Sie verlieren. Die Aufrufe von `print` und `println`, die dem Wort `else` unmittelbar folgen, sind ebenfalls Teil des Befehls `if`.) Genau genommen müssen Sie die Anweisungen innerhalb eines `if`-Befehls nicht einrücken. Wenn es um den Compiler geht, können Sie Ihren gesamten Code auf eine einzige Zeile oder in einem zwar kunstvollen, aber missgestalteten Zickzack schreiben. Das Problem ist dann nur, dass weder Sie noch irgendjemand sonst jemals in der Lage sein wird, in Ihrem Code einen Sinn zu erkennen. Der Grund dafür, den Code nach logischen Gesichtspunkten einzurücken, ist, ihn lesbar und damit verständlich zu machen. In [Listing 5.1](#) helfen die Einrückungen von `print` und `println` Ihren Augen dabei, schnell zu erkennen, dass diese Befehle dem `if/else`-Ablauf untergeordnet sind.

In einem kleinen Programm können fehlende oder nicht vernünftig ausgeführte Einrückungen gerade noch toleriert werden, aber in einem komplexen Programm sind fehlende oder schlecht ausgeführte Einrückungen ein Albtraum. Glücklicherweise verfügen viele IDEs über Werkzeuge, die den Code automatisch einrücken. Ich gehe sogar so weit zu behaupten, dass die Einrückung von Code zu den Funktionen einer IDE gehört, auf die ich sehr viel Wert lege.



Wenn Sie `if`-Anweisungen schreiben, könnten Sie in die Versuchung kommen, alle Regeln über geschweifte Klammern über Bord zu werfen und sich nur noch auf Einrückungen zu verlassen. Unglücklicherweise funktioniert dies nur selten. Wenn Sie nach dem Wort `else` drei Befehle einrücken und vergessen, sie mit geschweiften Klammern zu umgeben, glaubt der Computer, dass der `else`-Teil nur aus dem ersten der drei Befehle besteht. Schlimmer noch ist, dass die Einrückung Sie fälschlicherweise glauben lässt, dass der `else`-Teil alle drei Befehle umfasst. Dadurch wird es für Sie noch schwieriger herauszufinden, warum sich Ihr Code nicht so verhält, wie Sie geplant haben. Achten Sie also auf die Klammern!

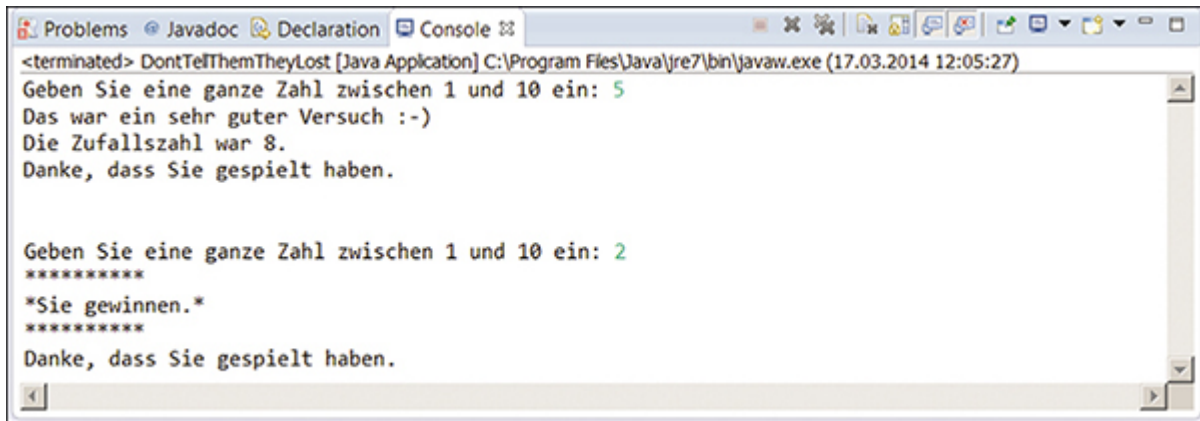
Eselos in Ifrika

Okay, der Titel dieses Abschnitts ist erfunden. Die Idee dahinter ist, dass Sie eine `if`-Anweisung erstellen können, die keinen `else`-Teil enthält. Vielleicht möchten Sie in [Listing 5.1](#) dem Benutzer nicht so deutlich sagen, dass er verloren hat. Deshalb habe ich in [Listing 5.2](#) den Code etwas abgewandelt.

```
import static java.lang.System.in;
import static java.lang.System.out;
import java.util.Scanner;
import java.util.Random;
public class DontTellThemTheyLost {
    public static void main(String args[]) {
        Scanner keyboard = new Scanner(in);
        out.print("Geben Sie eine ganze Zahl "
            + "zwischen 1 und 10 ein: ");
        int inputNumber = keyboard.nextInt();
        int randomNumber = new Random().nextInt(10) +
1;
        if (inputNumber == randomNumber) {
            out.println("*Sie gewinnen.*");
        }
        out.println("Das war ein sehr guter Versuch :
-)"");
        out.print("Die Zufallszahl war ");
        out.println(randomNumber + ".");
        out.println("Danke, dass Sie gespielt haben."
);
        keyboard.close();
    }
}
```

[Listing 5.2](#): Ein höflicheres Ratespiel

Die `if`-Anweisung in [Listing 5.2](#) hat keinen `else`-Teil. Wenn `inputNumber` mit `randomNumber` identisch ist, gibt der Computer Sie gewinnen aus, was er anderenfalls nicht macht.



```
<terminated> DontTellThemTheyLost [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (17.03.2014 12:05:27)
Geben Sie eine ganze Zahl zwischen 1 und 10 ein: 5
Das war ein sehr guter Versuch :-)
Die Zufallszahl war 8.
Danke, dass Sie gespielt haben.

Geben Sie eine ganze Zahl zwischen 1 und 10 ein: 2
*****
*Sie gewinnen.*
*****
Danke, dass Sie gespielt haben.
```

Abbildung 5.3: Zwei Ausführungen des Spiels in Listing 5.2

[Listing 5.2](#) enthält eine neue Idee. Wenn ich für `System.in` eine `import`-Deklaration erstelle, bin ich in der Lage, `new Scanner(System.in)` auf die Kurzform `new Scanner(in)` zu reduzieren. Offen gestanden erspart das Hinzufügen dieser `import`-Deklaration so gut wie keine Zeit. Letztendlich kostet es sogar mehr Schreiarbeit, mit dieser Deklaration zu arbeiten, als auf sie zu verzichten. Aber ich wollte Ihnen zeigen, dass es möglich ist, `System.in` zu importieren.

Bedingungen mit Vergleichsoperatoren und mit logischen Operatoren bilden

Java verfügt über viele kleine Gimmicks und Schmankerl, die es Ihnen ermöglichen, Bedingungen so zu formulieren, wie es Ihren Bedürfnissen entspricht. Dieser Abschnitt stellt Ihnen vor, was da alles auf Sie wartet.

Zahlen miteinander vergleichen: Vergleichsoperatoren

Tabelle 5.1 enthält die Operatoren, die Sie verwenden können, um Elemente miteinander zu vergleichen.

Symbol	Bedeutung	Beispiel
==	ist gleich	<code>anzahlKatzen == 5</code>
!=	ist ungleich	<code>anzahlKatzen != 5</code>
<	ist kleiner als	<code>anzahlKatzen < 5</code>
>	ist größer als	<code>anfangsBuchstabe > 'H'</code>
<=	ist kleiner oder gleich	<code>anzahlKatzen <= 5</code>
>=	ist größer oder gleich	<code>anzahlKatzen >= 5</code>

Tabelle 5.1: Vergleichsoperatoren

Sie können alle Vergleichsoperatoren verwenden, die es in Java gibt, um Zahlen und Zeichen miteinander zu vergleichen. Wenn es um Zahlen geht, läuft alles so ab, wie Sie es aus der Mathematik her kennen. Sobald Sie es aber mit Zeichen zu tun haben, verlangen die Vergleichsoperationen ein wenig Aufmerksamkeit von Ihnen. Wenn Sie Großbuchstaben miteinander vergleichen wollen, geht dies problemlos. Der Buchstabe *B* steht im Alphabet vor dem Buchstaben *H*, wodurch die Bedingung `'B' < 'H'` wahr ist. Auch der Vergleich von Kleinbuchstaben läuft ohne Schwierigkeiten ab. Zu Problemen kommt es erst, wenn Sie einen Großbuchstaben mit einem Kleinbuchstaben vergleichen wollen, weil der Großbuchstabe immer kleiner ist als der Kleinbuchstabe. Deshalb ist `'Z' < 'A'` zwar falsch, aber `'Z' < 'a'` wahr.



Die Buchstaben A bis Z werden numerisch durch die Zahlen 65 bis 90 dargestellt, während dies bei den Buchstaben a bis z durch die Zahlen 97 bis 122 erfolgt. Aus diesem Grund sind Kleinbuchstaben größer als Großbuchstaben.



Passen Sie auf, wenn Sie zwei Zahlen auf Gleichheit (mit `==`) oder Ungleichheit (mit `!=`) überprüfen. Wenn Sie Berechnungen durchgeführt und zwei `double`- oder `float`-Werte erhalten haben, sind diese Werte selten wirklich gleich. (Das Problem rührt von den lästigen Nachkommastellen her.)

So entsprechen zum Beispiel 21 Grad Celsius 69,8 Grad Fahrenheit. Wenn Sie die entsprechende Berechnung $(9,0 / 5 * 21 + 32)$ manuell durchführen, lautet das Ergebnis auch 69,8. Aber die Bedingung $9,0 / 5 * 21 + 32 == 69.8$ hat den Wert `false`, denn wenn der Computer $9,0 / 5 * 21 + 32$ berechnet, lautet das Ergebnis 69.800000000000001 und nicht 69.8.

Objekte vergleichen

Wenn Sie anfangen, mit Objekten zu arbeiten, werden Sie herausfinden, dass Sie auch Objekte mit `==` und `!=` vergleichen können. So ist zum Beispiel eine Schaltfläche, die Sie auf dem Bildschirm Ihres Computers sehen, ein Objekt. Und Sie können fragen, ob es sich bei dem Ding, das gerade mit der Maus angeklickt worden ist, um eine bestimmte Schaltfläche handelt. Diese Abfrage erledigen Sie mit Javas Gleichheitsoperator:

```
if (e.getSource() == bCopy) {  
    clipboard.setText(which.getText());  
}
```

Sie finden in [Kapitel 16](#) mehr über das Antwortverhalten von Schaltflächen heraus.

Zum großen Finale mit Javas Vergleichsoperatoren kommt es, wenn Sie zwei Strings (alphanumerische Zeichenfolgen) vergleichen. (Näheres über Strings steht in [Kapitel 4](#) im Abschnitt über Referenztypen.) Wenn Sie zwei Strings miteinander vergleichen, sollten Sie auf das doppelte Gleichheitszeichen verzichten. Mit dem doppelten Gleichheitszeichen fragen Sie: »Ist dieser String an genau derselben Stelle im Arbeitsspeicher abgelegt worden wie der andere String?« Normalerweise wollen Sie diese Frage gar nicht stellen. Stattdessen möchten Sie wissen: »Enthält dieser String dieselben Zeichen wie der andere String?« Um diese zweite Frage zu beantworten, enthält Javas `String`-Typ eine Methode, die `equals` heißt. (Diese Methode ist, wie alles im bekannten Universum, Bestandteil der Java-API.) Die Methode `equals` vergleicht zwei Strings (Zeichenfolgen) miteinander, um herauszufinden, ob sie aus denselben Zeichen bestehen. [Listing 5.3](#) enthält ein Beispiel für die Verwendung von

equals, und [Abbildung 5.4](#) zeigt, was das Programm in [Listing 5.3](#) ausgibt.

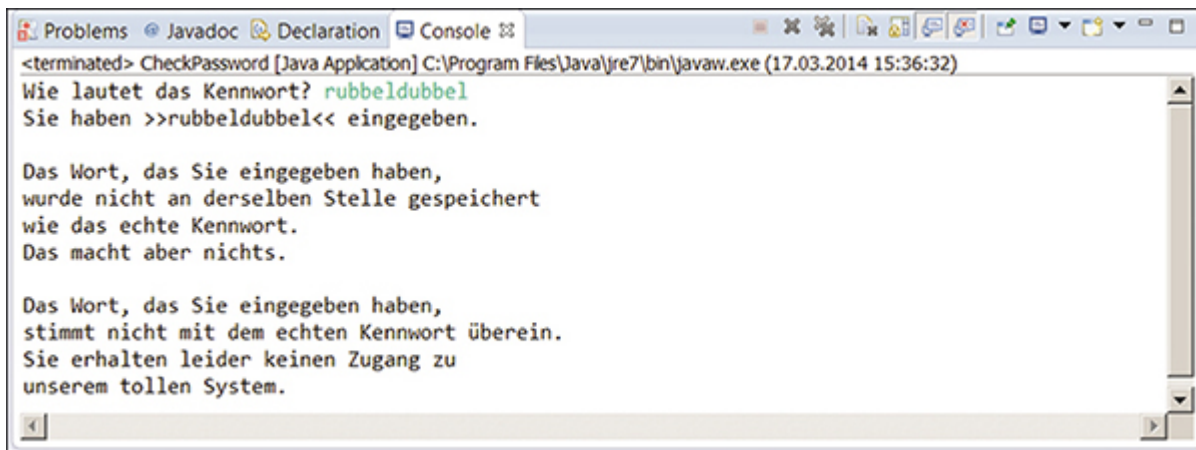
```
import static java.lang.System.*;
import java.util.Scanner;
public class CheckPassword {
    public static void main(String args[]) {
        out.print("Wie lautet das Kennwort? ");
        Scanner keyboard = new Scanner(in);
        String password = keyboard.next();
        out.println("Sie haben >>" + password
            + "<< eingegeben.");
        out.println();
        if (password == "swordfish") {
            out.print("Das Wort, das Sie eingegeben haben
,");
            out.println(
                "wurde an derselben Stelle gespeichert");
            out.println("wie das echte Kennwort.");
            out.println("Sie sind wohl ein Hacker.");
        } else {
            out.println(
                "Das Wort, das Sie eingegeben haben,");
            out.println(
                "wurde nicht an derselben Stelle gespeicher
t");
            out.println("wie das echte Kennwort.");
            out.println("Das macht aber nichts.");
        }
        out.println();
        if (password.equals("swordfish")) {
            out.println(
                "Das Wort, das Sie eingegeben haben,");
            out.println(
                "hat dieselben Zeichen wie das echte");
            out.println("Kennwort. Sie erhalten Zugang zu
");
            out.println("unserem tollen systemSystem.
");
        } else {
            out.println(
                "Das Wort, das Sie eingegeben haben,");
            out.println("stimmt nicht mit dem echten "
);
            out.println("Kennwort überein.");
        }
    }
}
```

```

        out.println("Sie erhalten leider keinen "
);
        out.println("Zugang zu unserem tollen "
+ "System.");
    }
    keyboard.close();
}
}

```

[Listing 5.3](#): Ein Kennwort überprüfen



[Abbildung 5.4](#): Ein Vergleich von == und der Methode *equals*

Sie gehen am besten so vor, dass Sie die Java-Methode `equals` verwenden. Diese Methode hat eine ungewöhnliche Syntax: Wenn Sie sie aufrufen, müssen Sie hinter einem der Strings einen Punkt setzen, während der andere String von Klammern eingeschlossen wird.

Für den Aufruf der Methode `equals` ist es ohne Bedeutung, welcher der Strings den Punkt und welcher die Klammern erhält. So hätten Sie in [Listing 3.3](#) auch

```
if ("swordfish".equals(password))
```

schreiben können. Diese Methode würde genauso gut funktionieren.



Ein Aufruf der Java-Methode `equals` sieht unsymmetrisch aus, ist es aber nicht. Hinter diesem scheinbaren

Missverhältnis aus Punkt und Klammern steckt als Grundgedanke, dass Sie es mit zwei Objekten zu tun haben: dem Objekt `password` und dem Objekt `"swordfish"`. Beide Objekte sind vom Typ `String`. (Genau genommen ist `password` eine Variable vom Typ `String` und `"swordfish"` ein `String-Literal`.) Wenn Sie schreiben `password.equals("swordfish")`, rufen Sie eine `equals`-Methode auf, die zum `password`-Objekt gehört, und übergeben ihr `"swordfish"` als Parameter. (Sie können in [Kapitel 7](#) mehr über Methoden erfahren, die zu Objekten gehören.)



Wenn Sie Strings miteinander vergleichen wollen, verwenden Sie die Methode `equals` und nicht das doppelte Gleichheitszeichen.

Alles auf einen Schlag importieren

Die erste Zeile von [Listing 5.3](#) zeigt einen bequemen Weg auf, um sowohl `System.out` als auch `System.in` zu importieren. Wenn Sie alles importieren möchten, was `System` zu bieten hat, verwenden Sie das Sternchen (*) als Platzhalter. Tatsächlich ersetzt `java.lang. System.*` 40 einzelne Importdeklarationen, zu denen auch `System.in`, `System.out`, `System.err`, `System.nanoTime` und andere `System`-Dinge gehören.

Im Allgemeinen ist es keine gute Programmierpraxis, in einer `import`-Deklaration ein Sternchen zu verwenden, weshalb ich in den Beispielen dieses Buches normalerweise darauf verzichte. Wenn es aber um größere Programme geht, die Dutzende von Namen aus der Java-API verwenden, ist der Trick mit dem Sternchen ganz schön praktisch.



Sie können ein Sternchen nicht irgendwo in einer `import`-Deklaration unterbringen. So lässt sich zum Beispiel nicht alles importieren, das mit `java` anfängt, indem Sie

`import java.*` schreiben. Sie dürfen ein Sternchen nur als Ersatz für einen Klassennamen oder für den Namen von etwas Statischem verwenden, das in einer Klasse untergebracht ist. Sie finden in [Kapitel 9](#) weitergehende Informationen für den Einsatz von Sternchen in `import`-Deklarationen; Statisches behandle ich in [Kapitel 10](#).

Javas logische Operatoren

Mr. Spock wäre zufrieden. Java verfügt über alle Operatoren, die logische Tests benötigen. Ich habe sie in [Tabelle 5.2](#) zusammengefasst.

Operator	Bedeutung	Beispiel
<code>&&</code>	Und	<code>5 < x && x < 10</code>
<code> </code>	oder	<code>x < 5 10 < x</code>
<code>!</code>	nicht	<code>!password.equals("swordfish")</code>

Tabelle 5.2: Logische Operatoren

Sie können mit diesen Operatoren umfangreiche Bedingungen formulieren. [Listing 5.4](#) enthält ein Beispiel.

```
import javax.swing.JOptionPane;
public class Authenticator {
    public static void main(String args[]) {
        String username =
            JOptionPane.showInputDialog("Benutzername
:");
        String password =
            JOptionPane.showInputDialog("Kennwort:");
        if (
            username != null &&
            password != null &&
            (
                (username.equals("bburd") &&
                 password.equals("swordfish")) ||
                (username.equals("hritter") &&
                 password.equals("preakston"))
            )
        )
    }
}
```

```

        JOptionPane.showMessageDialog
            (null, "Sie erhalten Zugang.");
    } else {
        JOptionPane.showMessageDialog
            (null, "Sie benehmen sich verdächtig.
");
    }
}
}

```

[Listing 5.4](#): Benutzernamen und Kennwort überprüfen

[Abbildung 5.5](#) zeigt zwei Läufe des Programms in Listing 5.4. Ähnlich erfolgreich wie mit `bbrud` und `swordfish` sind Sie mit dem Benutzernamen `hritter` und dem Kennwort `preakston`. Alle anderen Kombinationen – einschließlich leerer Felder – scheitern.

Ich muss gestehen, dass [Abbildung 5.5](#) auf einer Täuschung beruht. Damit Sie den Benutzernamen und das entsprechende Kennwort lesen können, habe ich [Listing 5.4](#) um einen zusätzlichen Befehl erweitert. Diese Anweisung (`UIManager.put("TextField.font", new Font("Dialog", Font.BOLD, 14))`) vergrößert die Schriftart in den Textfeldern.

[Listing 5.4](#) demonstriert aber auch eine neue Vorgehensweise, um an die Eingaben eines Benutzers zu gelangen, wobei es hier primär darum geht, dem Benutzer für die Eingabe Dialogfelder vorzusetzen. Der Befehl

```

String password =
    JOptionPane.showInputDialog("Kennwort:");
erledigt mehr oder weniger dieselbe Aufgabe wie
String password = keyboard.next();

```

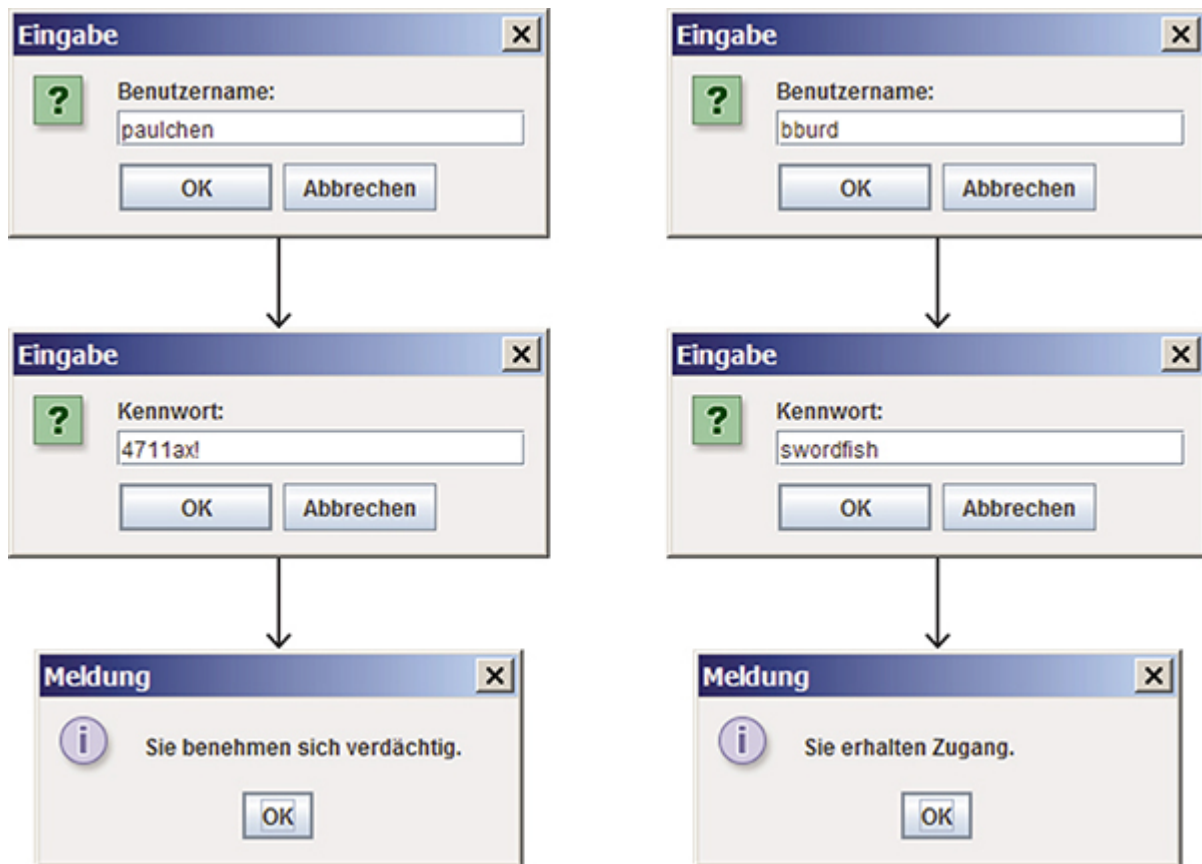


Abbildung 5.5: Zwei Ausführungen des Codes in [Listing 5.4](#)

in [Listing 5.3](#). Der große Unterschied liegt darin, dass `keyboard.next()` in der CONSOLE einen nicht sonderlich ansprechenden Text anzeigt, während `JOptionPane.showInputDialog("Benutzername:")` ein chic aussehendes Dialogfeld ausgibt, das ein Textfeld und eine Schaltfläche enthält. (Vergleichen Sie [Abbildung 5.4](#) und [Abbildung 5.5](#) miteinander.) Wenn der Benutzer auf OK klickt, übernimmt der Computer den Text, der sich im Textfeld befindet, und übergibt ihn an eine Variable. Tatsächlich ist es so, dass [Listing 5.4](#) `JOptionPane.showInputDialog` zweimal verwendet – einmal, um einen Benutzernamen zu erhalten, und ein zweites Mal, damit der Benutzer einen Wert für die Variable `password` eingeben kann.

Fast am Ende von [Listing 5.4](#) verwende ich eine leichte Variation von `JOptionPane`:

```
JOptionPane.showMessageDialog  
    (null, "Sie erhalten Zugang.");
```

Mit `showMessageDialog` zeige ich ein ganz einfaches Dialogfeld an – ein Dialogfeld ohne Textfeld (siehe noch einmal [Abbildung 5.5](#)).

Der Name `JOptionPane` ist, wie Tausende anderer Namen auch, in Javas API definiert worden. (Um genau zu sein: Die Quelle von `JOptionPane` heißt `javax.swing`, was wiederum innerhalb von Javas API definiert wird.) Deshalb habe ich in [Listing 5.4](#) `javax.swing.JOptionPane` importiert, damit ich den Namen `JOptionPane` verwenden kann. Ich importiere `javax.swing.JOptionPane` ganz zu Beginn des Listings.



`JOptionPane.showInputDialog` funktioniert in [Listing 5.4](#) deshalb gut, weil die Eingabe des Benutzers nur aus Zeichen besteht (Benutzername und Kennwort), die nicht zum Rechnen verwendet werden. Wenn Sie wollen, dass der Benutzer Zahlen (zum Beispiel `int` oder `double`) eingibt, müssen Sie zusätzliche Arbeit investieren. Um vom Benutzer einen `int`-Wert zu erhalten, schreiben Sie zum Beispiel `int numberOfCows = Integer.parseInt(JOptionPane.showInputDialog("Wie viele Kühe?"))`. Das zusätzliche `Integer.parseInt` erzwingt die Eingabe eines `int`-Wertes. Um vom Benutzer einen `double`-Wert zu erhalten, schreiben Sie so etwas wie `perMilleOfAlcohol = Double.parseDouble(JOptionPane.showInputDialog("Promille:"))`. Dabei sorgt `Double.parseDouble` dafür, dass es sich bei der Eingabe in das Textfeld um einen `double`-Wert handeln muss.

Vive les nuls!

In der französischen Übersetzung heißen die »... für Dummies«-Bücher »... Pour les Nuls«. Also ist ein deutscher *Dummy* in

Frankreich eine *Null*. In Java bedeutet `null` *nichts* oder *leer*.
Wenn Sie in Listing 5.4

```
if (
    username != null
    lesen, bedeutet dies nichts anderes als
```

```
if (
    username ist leer
oder
```

```
if (
    username hat einen beliebigen Wert
```

Um herauszufinden, wie es passieren kann, dass `username` keinen Wert hat, sollten Sie sich noch einmal [Abbildung 5.5](#) anschauen. Wenn Sie im ersten Dialogfeld auf ABBRECHEN klicken, übergibt der Computer `null` an Ihr Programm. Damit erhält in [Listing 5.4](#) die Variable `username` den Wert `null`. Der Vergleich von `username != null` prüft nach, ob der Benutzer im ersten Dialogfeld des Programms auf ABBRECHEN geklickt hat. Der Vergleich `password != null` führt im zweiten Dialogfeld eine ähnliche Prüfung durch. Wenn Sie sich in [Listing 5.4](#) mit der `if`-Anweisung beschäftigen, lesen Sie eigentlich dieses hier:

```
wenn (if) (
    Sie im Dialogfeld für die Eingabe des Benutzernam
ens
    nicht auf Abbrechen geklickt und
    Sie im Dialogfeld für die Eingabe des Kennwortes
    nicht auf Abbrechen geklickt haben und
    (
        (Sie »bburd« im Feld für den Benutzernamen) u
nd
        Sie "swordfish" als Kennwort eingegeben habe
n)
    oder
    (Sie »hritter« im Feld für den Benutzernamen
und
        Sie »preakston« im Feld für das Kennwort
        eingegeben haben)
    )
)
```



In [Listing 5.4](#) sind die beiden Vergleichsaktionen `username != null` und `password != null` verpflichtend. Wenn Sie diese Befehle vergessen und während der Ausführung des Programms auf **ABBRECHEN** klicken, erhalten Sie eine üble `NullPointerException`-Meldung, und das Programm stürzt vor Ihren Augen ab. Das Wort `null` steht in Java für *nichts*, und in dieser Programmiersprache ist es einfach unmöglich, einen String wie `"bburd"` oder `"swordfish"` mit *nichts* zu vergleichen. In [Listing 5.4](#) sorgt der Vergleich von `username != null` dafür, dass Java mit der Prüfung von `username.equals("bburd")` nicht weitermacht, wenn der Benutzer auf **ABBRECHEN** klickt. Ohne den Einstiegstest `username != null` fordern Sie Probleme heraus.



Das letzte Pärchen `null`en in [Listing 5.4](#) unterscheidet sich von den anderen. Im Code `JOptionPane.showMessageDialog (null, "Sie erhalten Zugang.")` steht `null` für »kein anderes Dialogfeld«. Im Einzelnen: Der Aufruf von `showMessageDialog` weist Java an, ein neues Dialogfeld erscheinen zu lassen, und das Wort `null` gibt an, dass das neue Dialogfeld nicht aus einem vorhandenen Dialogfeld heraus entstehen soll. Merken Sie sich, dass Java darauf besteht, dass Sie etwas dazu sagen, wo der Ursprung des neuen Dialogfeldes liegt. (Aus irgendeinem Grund verzichtet Java darauf, dass Sie für `showInputDialog` einen Ursprung angeben müssen. Da soll mal einer schlau draus werden!) Auf jeden Fall ist es richtig gut, in [Listing 5.4](#) über `showMessageDialog` auf Dialogfelder zugreifen zu können, die aus dem Nichts auftauchen.

(Bedingungen in Klammern)

Behalten Sie die Klammern im Auge! Wenn Sie Bedingungen mit logischen Operatoren zusammenfügen, lohnt es sich, damit Zeit

zu »vergeuden«, Klammern zu setzen, die nicht unbedingt notwendig sind. Das ist allemal besser, als die Ergebnisse Ihrer Arbeit dadurch zu vermässeln, dass Sie irgendwo auf Klammern verzichten. Nehmen Sie zum Beispiel den Ausdruck:

```
2 < 5 || 100 < 6 && 27 < 1
```

Wenn Sie diesen Ausdruck falsch lesen, könnten Sie zu dem Ergebnis kommen, dass der Wert des Ausdrucks `false` ist. Sie könnten nämlich fälschlicherweise glauben, dass der Ausdruck *(etwas oder etwas anderes) && 27 < 1* bedeutet. Da nun `27 < 1` falsch ist, schließen Sie, dass der ganze Ausdruck falsch ist. In Wirklichkeit wird nun aber der Operator `&&` vor dem Operator `||` verarbeitet. Der Ausdruck fragt also tatsächlich `2 < 5 || (etwas oder etwas anderes)`. Da `2 < 5` wahr ist, ist der ganze Ausdruck wahr.

Um den Wert des Ausdrucks von `true` in `false` zu ändern, umgeben Sie die ersten beiden Vergleiche des Ausdrucks mit einer Klammer:

```
(2 < 5 || 100 < 6) && 27 < 1
```



Der Java-Operator `||` ist *inklusiv*. Das bedeutet, dass Sie als Wert `true` erhalten, wenn der Operand auf der linken Seite oder der Operand auf der rechten Seite oder beide Seiten des Ausdrucks wahr ist beziehungsweise sind. So ist zum Beispiel der Ausdruck `2 < 10 || 20 < 30` wahr.



In Java ist es nicht möglich, Vergleiche wie in der Umgangssprache zu kombinieren. Im Alltag können Sie sagen: »Wir erwarten zum Abendessen zwischen drei und zehn Personen.« In Java erhalten Sie eine Fehlermeldung, wenn Sie schreiben `3 <= personen <= 10`. Um einen solchen Vergleich hinzubekommen, benötigen Sie etwas wie `3 <= personen && personen <= 10`.

In [Listing 5.4](#) enthält der `if`-Befehl mehr als ein Dutzend Klammern. Was passiert, wenn Sie zwei davon weglassen?

```

if (
    username != null &&
    password != null &&
    // die öffnende Klammer wird weggelassen
    (username.equals("bburd") &&
     password.equals("swordfish")) ||
    (username.equals("hritter") &&
     password.equals("preakston"))
    // die schließende Klammer wird weggelassen
)

```

Java versucht, Ihre Wünsche zu deuten, indem es alles vor dem »oder« (dem Operator `||`) gruppiert:

```

if (
    username != null &&
    password != null &&
    (username.equals("bburd") &&
     password.equals("swordfish"))
    ||
    (username.equals("hritter") &&
     password.equals("preakston"))
)

```

Wenn der Benutzer auf ABBRECHEN klickt und `username` den Wert `null` hat, sagt Java: »Okay! Der Kram vor dem Operator `||` ist falsch, aber vielleicht gilt das nicht für das, was hinter diesem Operator steht. Ich werde also die Informationen dort prüfen.« (Java führt häufig Selbstgespräche und befindet sich deshalb auch in psychiatrischer Behandlung.)

Sei's drum, aber wenn Java letztendlich `username.equals("hritter")` überprüft, bricht Ihr Programm mit einer hässlichen Meldung vom Typ `NullPointerException` ab. Sie haben Java dadurch verärgert, dass Sie versucht haben, `equals` bei einem leeren `username` anzuwenden.

Verschachtelungen

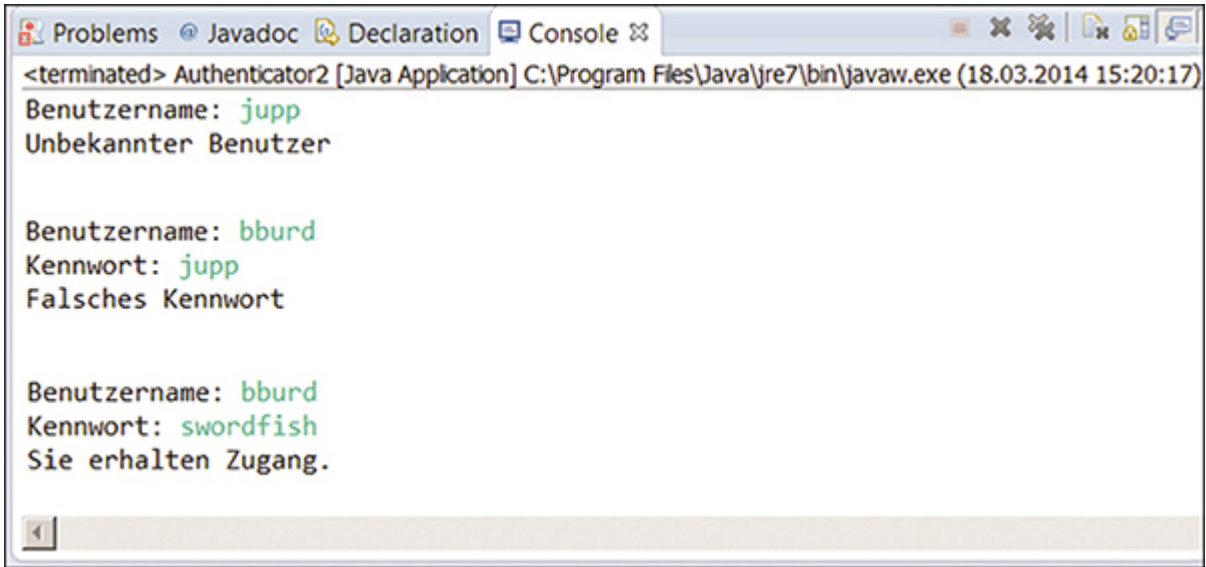
Kennen Sie die ineinander verschachtelten russischen Puppen, die *Matroschkas*? Wenn Sie eine Puppe öffnen, finden Sie darin eine zweite Puppe, in der es eine dritte und darin eine vierte gibt.

Dasselbe können Sie mit Javas `if`-Anweisungen machen. [Listing 5.5](#) zeigt Ihnen, wie das geht.

```
import static java.lang.System.out;
import java.util.Scanner;
public class Authenticator2 {
    public static void main(String args[]) {
        Scanner keyboard = new Scanner(System.in);
        out.print("Benutzername: ");
        String username = keyboard.next();
        if (username.equals("bburd")) {
            out.print("Kennwort: ");
            String password = keyboard.next();
            if (password.equals("swordfish")) {
                out.println("Sie erhalten Zugang.");
            } else {
                out.println("Falsches Kennwort");
            }
        } else {
            out.println("Unbekannter Benutzer");
        }
        keyboard.close();
    }
}
```

[Listing 5.5](#): Verschachtelte *if*-Anweisungen

[Abbildung 5.6](#) gibt verschiedene Läufe des Codes in [Listing 5.5](#) wieder. Die Grundidee ist, dass Sie zwei Tests bestehen müssen, um sich anzumelden (oder anders ausgedrückt, zwei Bedingungen müssen wahr sein). Die erste Bedingung prüft, ob der Benutzername gültig ist; die zweite Bedingung prüft, ob das Kennwort richtig ist. Wenn Sie die erste Prüfung überstehen (die auf den Benutzernamen), gelangen Sie sofort zu einer weiteren `if`-Anweisung, die einen zweiten Test durchführt (den auf das Kennwort). Falls Sie den ersten Test nicht bestehen, kommen Sie gar nicht erst zum zweiten. [Abbildung 5.7](#) bietet einen Überblick über den allgemeinen Ablauf.



```
<terminated> Authenticator2 [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (18.03.2014 15:20:17)
Benutzername: jupp
Unbekannter Benutzer

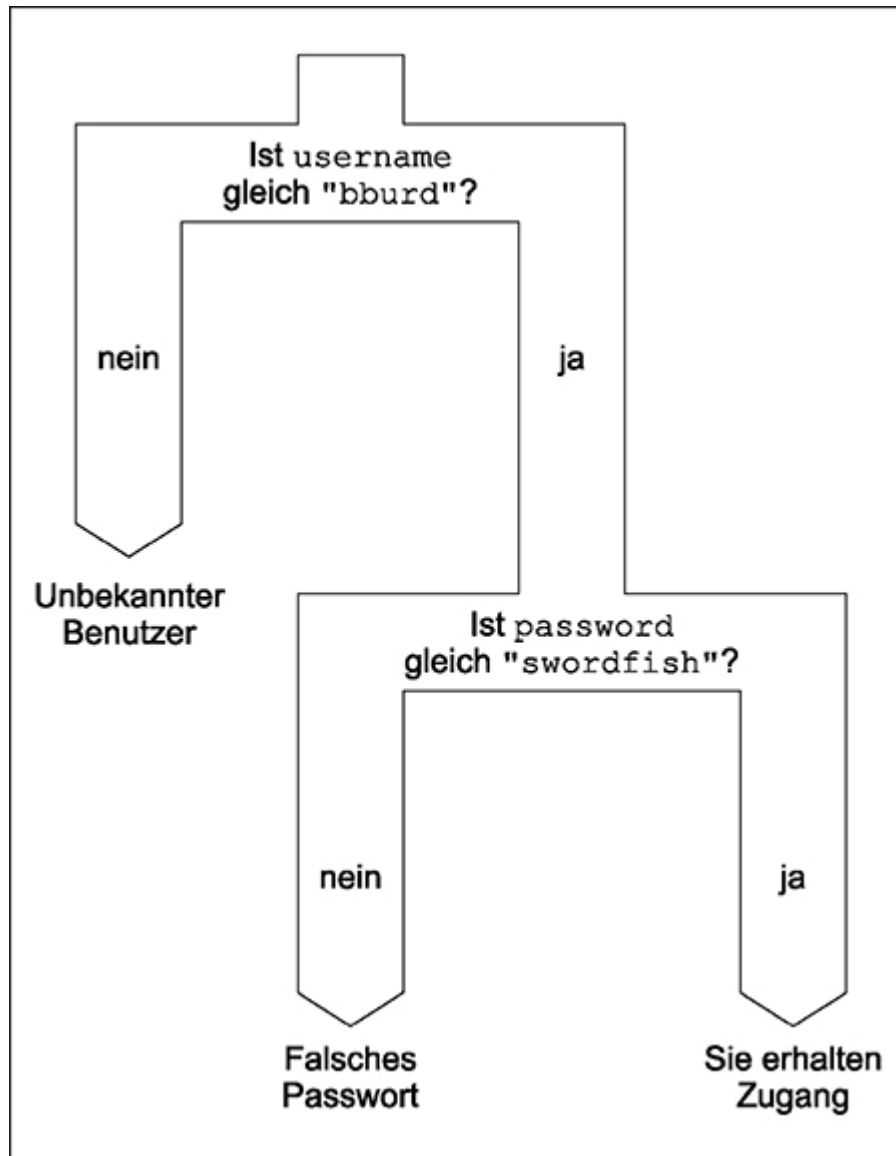
Benutzername: bburd
Kennwort: jupp
Falsches Kennwort

Benutzername: bburd
Kennwort: swordfish
Sie erhalten Zugang.
```

Abbildung 5.6: Das Programm in [Listing 5.5](#) wurde dreimal ausgeführt.



Der Code in [Listing 5.5](#) erledigt seinen Job mit verschachtelten `if`-Anweisungen wirklich gut, aber wenn es um eine echte Benutzerauthentifizierung geht, sollte Sie diesen Code besser vergessen. Erstens dürfen Sie ein Kennwort niemals in Klarschrift anzeigen (verwenden Sie immer Sternchen, um das Kennwort zu maskieren). Zweitens müssen Sie Kennwörter verschlüsseln. Drittens dürfen Sie einem böartigen Benutzer nie bekannt geben, welches der beiden Wörter (der Benutzername oder das Kennwort) falsch eingegeben wurde. Und viertens ... ich könnte noch eine Weile weitermachen. Der Code in [Listing 5.5](#) ist nicht dafür gedacht, als praktisches Beispiel für den Umgang mit dem Paar Benutzername/Kennwort zu dienen.



[Abbildung 5.7](#): Versuchen Sie nicht, mit dieser Gabel zu essen.

Unter mehreren Alternativen wählen (Javas »switch«-Befehle)

Ich bin der Erste, der zugibt, dass er es hasst, Entscheidungen zu fällen. Wenn etwas schief läuft, wünsche ich mir, dass jemand anders dafür verantwortlich ist. Das Schreiben des letzten

Abschnitts (über die Entscheidungsfindung mit Javas `if`-Anweisung) hat mich ziemlich fertiggemacht – was auch der Grund dafür ist, dass mein Geist ein wenig Unwillen zeigte, als ich diesen Abschnitt mit seinen Auswahlmöglichkeiten begann. Aber es tut gut, so etwas einzugestehen.

Der »switch«-Befehl

Nun ist es an der Zeit, dass Sie sich mit Situationen beschäftigen, in denen es eine Entscheidung mit vielen Alternativen gibt. Nehmen Sie den am Lagerfeuer so beliebten Song *Al's All Wet* (dessen vollständigen Text Sie im Kasten lesen können). Sie möchten unbedingt Code schreiben, der die Verse (oder Strophen) dieses Songs ausgibt. Glücklicherweise müssen Sie nicht alle Wörter immer wieder eingeben; Sie können die Wiederholungen im Text zu Ihrem Vorteil nutzen.

Al's All Wet

Gesungen zur Melodie von »Gentile Alouette«

Al's all wet. Oh, why is Al all wet? Oh, Al's all wet 'cause he's standing in the rain. Why is Al out in the rain? That's because he has no brain. Has no brain, has no brain, In the rain, in the rain. Ohhhhhhhh...

Al's all wet. Oh, why is Al all wet? Oh, Al's all wet 'cause he's standing in the rain. Why is Al out in the rain? That's because he is a pain. He's a pain, he's a pain, Has no brain, has no brain, In the rain, in the rain. Ohhhhhhhh...

Al's all wet. Oh, why is Al all wet? Oh, Al's all wet 'cause he's standing in the rain. Why is Al out in the rain? 'Cause this is the last refrain. Last refrain, last refrain, He's a pain, he's a pain, In the rain, in the rain. Ohhhhhhhh...

Al's all wet. Oh, why is Al all wet? Oh, Al's all wet 'cause he's standing in the rain.

– *Harriet Ritter and Barry Burd*

Ein komplettes Programm, um den gesamten Text von *Al's All Wet* anzuzeigen, wird erst in [Kapitel 6](#) vorgestellt. Bis dahin wollen wir davon ausgehen, dass Sie eine Variable mit dem Namen `verse` (deutsch *Vers*, *Strophe*) angelegt haben. Der Wert

von `verse` beträgt 1, 2, 3 oder 4, wobei dieser Wert davon abhängt, welche Strophe von *Al's All Wet* Sie ausgeben wollen. Sie könnten Code mit umständlichen `if`-Anweisungen schreiben, die alle denkbaren Strophennummern prüfen.

```
if (verse == 1) {
    out.println("That's because he has no brain.");
}
if (verse == 2) {
    out.println("That's because he is a pain.");
}
if (verse == 3) {
    out.println("'Cause this is the last refrain.");
}
```

Aber diese Vorgehensweise sieht recht unwirtschaftlich aus. Warum sollten Sie keinen Befehl erstellen, der den Wert von `verse` nur einmal prüft und dann die passende Aktion auf der Grundlage des gefundenen Wertes ausführt? So einen Befehl gibt es. Er heißt `switch`. [Listing 5.6](#) enthält ein Beispiel für den `switch`-Befehl.

```
import static java.lang.System.out;
import java.util.Scanner;
public class JustSwitchIt {
    public static void main(String args[]) {
        Scanner keyboard = new Scanner(System.in);
        out.print("Welcher Vers? ");
        int verse = keyboard.nextInt();
        switch (verse) {
            case 1:
                out.println("That
's because he has no brain.");
                break;
            case 2:
                out.println("That
's because he is a pain.");
                break;
            case 3:
                out.println("'Cause this is the last refrai
n.");
                break;
            default:
                out.println("Einen solchen Vers gibt es nic
ht. ");
        }
    }
}
```

```

        + "Bitte erneut versuchen.");
        break;
    }
    out.println("Ohhhhhhhh...");
    keyboard.close();
}
}

```

[Listing 5.6](#): Ein *switch*-Befehl

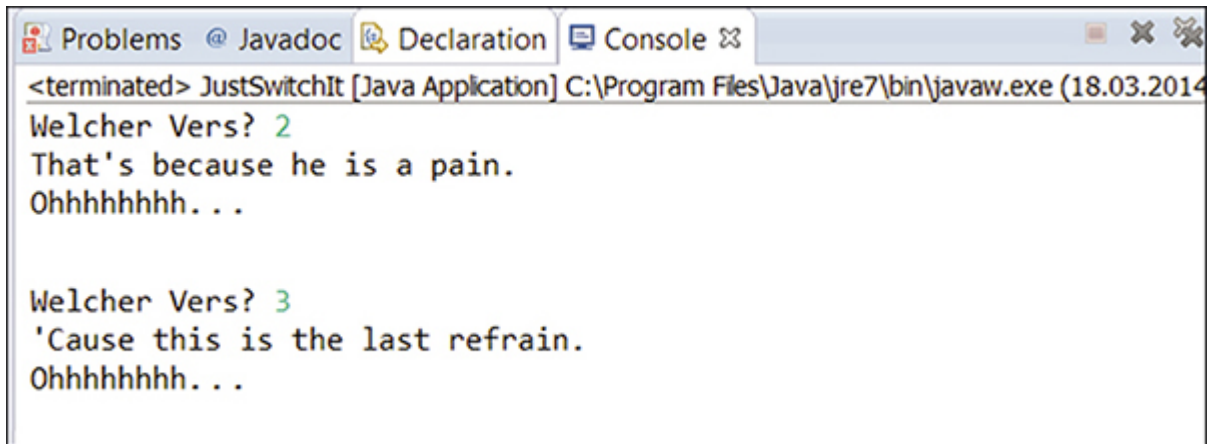
[Abbildung 5.8](#) zeigt das Ergebnis von zwei Läufen des Programms in [Listing 5.6](#). ([Abbildung 5.9](#) stellt die Struktur des Programms dar.) Zunächst gibt der Benutzer eine Zahl ein, beispielsweise 2. Dann erreicht die Ausführung des Programms den *switch*-Befehl. Der Computer prüft den Wert der Variablen *verse*. Wenn der Computer feststellt, dass dieser Wert 2 ist, überprüft er alle *case*-Klauseln des *switch*-Befehls. Der Wert 2 passt nicht zum obersten *case*, weshalb der Computer mit der mittleren der drei *case*-Klauseln weitermacht. Der Wert von *case* (die Zahl 2) stimmt mit dem Wert der Variablen *verse* überein, weshalb der Computer die Anweisungen ausführt, die unmittelbar auf *case 2* folgen. Dies sind

```

out.println("That's because he is a pain.");
break;

```

Der erste dieser beiden Befehle zeigt auf dem Bildschirm die Zeile *That's because he is a pain.* an. Der zweite Befehl ist ein *break*-Befehl. Wenn der Computer auf einen *break*-Befehl trifft, springt er aus der *switch*-Anweisung heraus, in der er sich gerade befindet. In [Listing 5.6](#) geschieht dies, unmittelbar nachdem er *'Cause this is the last refrain.* angezeigt hat. Der Computer verlässt den kompletten *switch*-Befehl und begibt sich direkt zu der Anweisung, die auf das Ende des *switch*-Befehls folgt. Er zeigt nun *Ohhhhhhhh...* an, weil dies die Anweisung verlangt, die auf den *switch*-Befehl folgt.



```
<terminated> JustSwitchIt [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (18.03.2014)
Welcher Vers? 2
That's because he is a pain.
Ohhhhhhhh...

Welcher Vers? 3
'Cause this is the last refrain.
Ohhhhhhhh...
```

[Abbildung 5.8](#): Zwei Ausführungen des Codes in Listing 5.6

Wenn ein lästiger Benutzer nach Vers 6 fragt, reagiert der Computer damit, dass er `case 1`, `case 2` und `case 3` überspringt und `default` ausführt und Einen solchen Vers gibt es nicht. Bitte erneut versuchen. anzeigt. Danach verlässt der Computer die `switch`-Anweisung endgültig und gibt `Ohhhhhhhh... auf dem Bildschirm aus.`



Es ist nicht zwingend notwendig, an das Ende eines `switch`-Befehls ein `break` zu setzen. In [Listing 5.6](#) gibt es das letzte `break` (das Teil von `default` ist) nur der Ordnung halber.

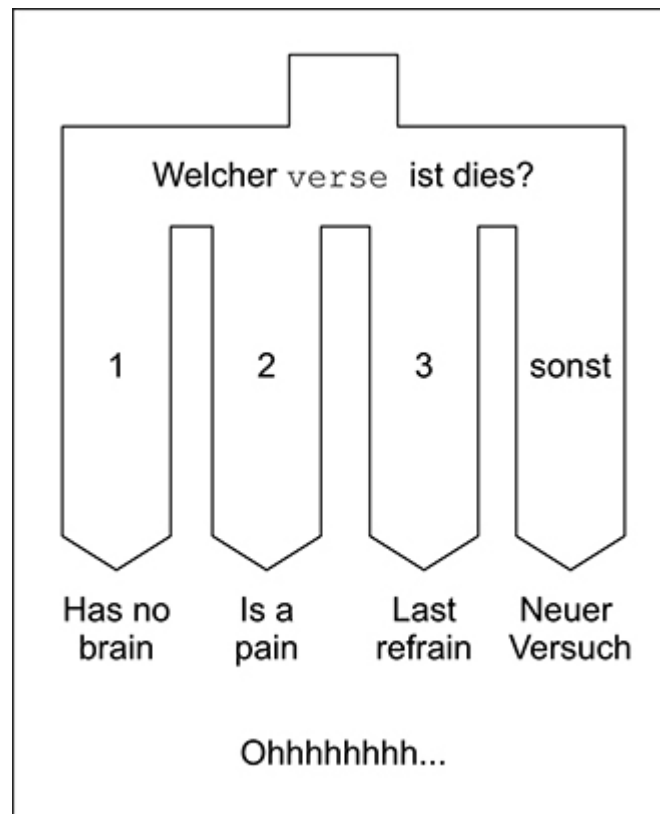


Abbildung 5.9: Die Struktur des `switch`-Befehls in [Listing 5.6](#)

To break or not to break

Im Leben eines jeden Java-Programmierers kommt irgendwann einmal der Zeitpunkt, an dem er einen `break`-Befehl vergisst. Zunächst sieht das, was dann auf dem Bildschirm ausgegeben wird, verwirrend aus, aber schon bald erinnert sich der Programmierer an ein Durchfallen. Der Begriff *Durchfallen* beschreibt, was geschieht, wenn Sie eine `case`-Klausel ohne `break` beenden. Das Programm fällt dann gewissermaßen bis zum nächsten `case` durch. Dieses Durchfallen wird so lange fortgesetzt, bis entweder ein `break`-Befehl oder das Ende des gesamten `switch`-Befehls erreicht wird.

Normalerweise möchten Sie natürlich bei einem `switch`-Befehl nicht durchfallen, weshalb Sie an den entsprechenden Stellen `break`-Befehle anbringen. Aber es gibt auch Situationen, in denen ein Durchfallen notwendig ist. Nehmen Sie als Beispiel den Song *Al's All Wet* (dessen Text im Kasten weiter vorn in

diesem Kapitel abgedruckt ist). Jeder Vers von *Al's All Wet* erweitert die Zeilen der vorherigen Strophe und erweitert diese um eine neue Zeile. Diese Situation (das Anwachsen der Zeilen von einem Vers zum nächsten) schreit regelrecht nach einem `switch`-Befehl mit Durchfallen. [Listing 5.7](#) setzt diese Idee um.

```
import static java.lang.System.out;
import java.util.Scanner;
public class FallingForYou {
    public static void main(String args[]) {
        Scanner keyboard = new Scanner(System.in);
        out.print("Welcher Vers? ");
        int verse = keyboard.nextInt();
        switch (verse) {
            case 3:
                out.print("Last refrain, ");
                out.println("last refrain,");
            case 2:
                out.print("He's a pain, ");
                out.println("he's a pain,");
            case 1:
                out.print("Has no brain, ");
                out.println("has no brain,");
        }
        out.println("In the rain, in the rain.");
        out.println("Ohhhhhhhh...");
        out.println();
        keyboard.close();
    }
}
```

[Listing 5.7](#): Ein *switch*-Befehl mit Durchfallen

[Abbildung 5.10](#) zeigt verschiedene Läufe des Programms in [Listing 5.7](#). Da der `switch`-Befehl keine `break`-Anweisungen enthält, fällt das Programm immer durch. Wenn der Benutzer beispielsweise Vers 2 auswählt, führt der Computer die beiden Anweisungen in `case 2` aus.

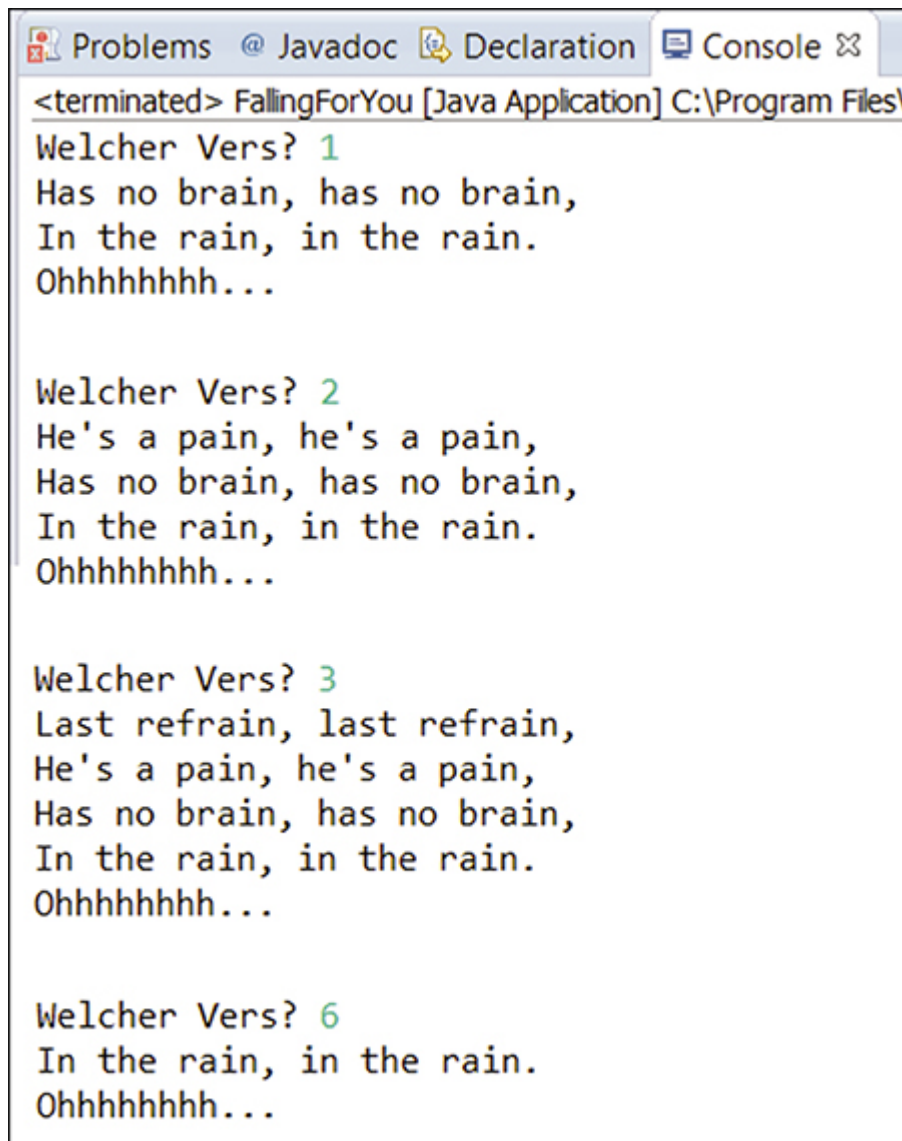
```
out.print("He's a pain, ");
out.println("he's a pain,");
```

Dann macht der Computer sofort mit der Ausführung der beiden Befehle in `case 1` weiter:

```
out.print("Has no brain, ");  
out.println("has no brain,");
```

Das ist auch richtig so, weil die zweite Strophe des Liedes alle Zeilen der ersten Strophe enthält.

Beachten Sie, was passiert, wenn der Benutzer nach Vers 6 fragt. Im `switch`-Befehl gibt es weder einen `case 6` noch ein `default`. Aus diesem Grund wird keine der Aktionen im `switch`-Befehl ausgeführt. und der Computer springt sofort zum ersten Befehl, der auf die `switch`-Anweisung folgt, und zeigt
In the rain, in the rain und Ohhhhhhhh... an.



```
<terminated> FallingForYou [Java Application] C:\Program Files\  
Welcher Vers? 1  
Has no brain, has no brain,  
In the rain, in the rain.  
Ohhhhhhhh...  
  
Welcher Vers? 2  
He's a pain, he's a pain,  
Has no brain, has no brain,  
In the rain, in the rain.  
Ohhhhhhhh...  
  
Welcher Vers? 3  
Last refrain, last refrain,  
He's a pain, he's a pain,  
Has no brain, has no brain,  
In the rain, in the rain.  
Ohhhhhhhh...  
  
Welcher Vers? 6  
In the rain, in the rain.  
Ohhhhhhhh...
```

[Abbildung 5.10](#): Vier Ausführungen des Programms in [Listing 5.7](#)

Der neue, verbesserte »switch«-Befehl

In den [Listing 5.6](#) und [5.7](#) steuert die Variable `verse` (ein `int`-Wert) die `switch`-Anweisung zu einem passenden `case`. Ein `int`-Wert kann in jeder Java-Version für `switch`-Befehle verwendet werden. (Da wir gerade dabei sind: Auch `char`-Werte und einige andere Wertarten können seit jeher in `switch`-Befehlen der Programmiersprache Java verwendet werden.)

Wenn Sie mit Java 7 oder später arbeiten, lässt es sich einrichten, dass der `case`, der in einem `switch`-Befehl ausgeführt werden soll, von dem Wert eines bestimmten Strings abhängt. [Listing 5.8](#) stellt den Einsatz von Strings in `switch`-Befehlen dar. [Abbildung 5.11](#) zeigt, was der Code ausgibt, wenn er ausgeführt wird.

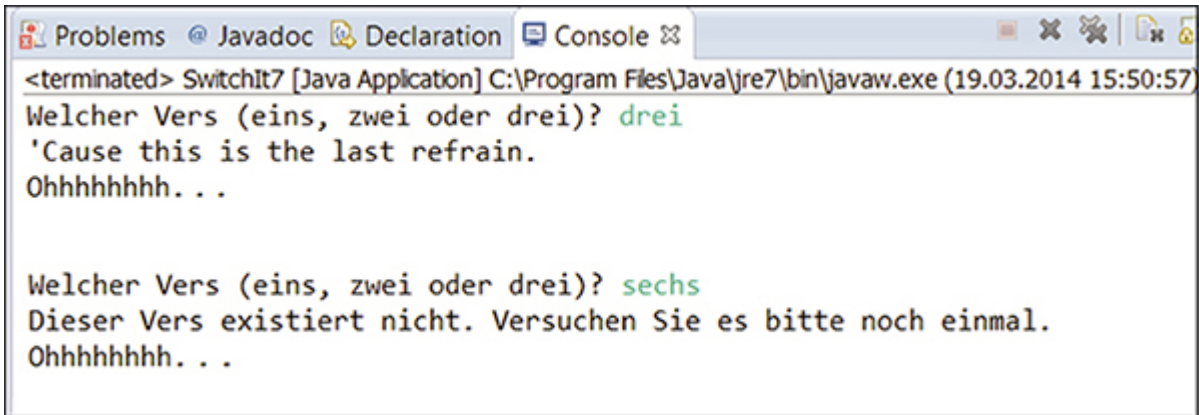
```
import static java.lang.System.out;
import java.util.Scanner;
public class SwitchIt7 {
    public static void main(String args[]) {
        Scanner keyboard = new Scanner(System.in);
        out.print("Welcher Vers (eins, zwei oder drei
? ");
        String verse = keyboard.next();
        switch (verse) {
            case "eins":
                out.println("That
's because he has no brain.");
                break;
            case "zwei":
                out.println("That
's because he is a pain.");
                break;
            case "drei":
                out.println("'Cause this is the last refrai
n.");
                break;
            default:
                out.println("Dieser Vers existiert nicht. "
+ "Versuchen Sie es bitte noch einmal.");
        }
    }
}
```

```

        break;
    }
    out.println("Ohhhhhhhh...");
    keyboard.close();
}
}

```

[Listing 5.8](#): Ein *switch*-Befehl mit einem String



```

<terminated> SwitchIt7 [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (19.03.2014 15:50:57)
Welcher Vers (eins, zwei oder drei)? drei
'Cause this is the last refrain.
Ohhhhhhhh...

Welcher Vers (eins, zwei oder drei)? sechs
Dieser Vers existiert nicht. Versuchen Sie es bitte noch einmal.
Ohhhhhhhh...

```

[Abbildung 5.11](#): Zwei Läufe des Programms in Listing 5.8



Wenn Sie Java verwenden, um Android-Apps zu erstellen, müssen Sie auf Java 5, Java 6 oder eine ältere Java-Version zurückgreifen, und Sie können keinen String das Schicksal eines `switch`-Befehls entscheiden lassen.

6

Den Programmablauf mit Schleifen steuern

In diesem Kapitel

- ▶ Schleifen verwenden
 - ▶ Schleifen zählen
 - ▶ Erbarmungslos wiederholen (bis der Benutzer eine klare Antwort gibt)
-

1966 schrieb das Unternehmen Geschichte, dem wir Head & Shoulders zu verdanken haben. Sie konnten auf der Rückseite der Flasche als Gebrauchsanleitung lesen: »EINSCHÄUMEN, AUSWASCHEN, WIEDERHOLEN.« Niemals zuvor ist eine Gebrauchsanleitung so erfolgreich zusammengefasst worden. Die Leute, die sich beruflich um Gebrauchsanleitungen kümmern, betrachteten diese Botschaft als wegweisende Leistung. Anleitungen wie diese unterschieden sich stark von dem, was damals alltäglich war. (So lautete zum Beispiel der erste Satz auf einer Dose mit Mückenspray: »Drehen Sie diese Dose so, dass Sie von Ihrem Gesicht wegzeigt.«)

Neben ihrer Kürze bestand das Besondere, das die Anleitung von Head & Shoulders so cool machte, aus drei einfachen Wörtern, die es schafften, eine Idee einzufangen, die die Kernaussage aller Anleitungen bildet: Wiederholung. Das letzte Wort, *wiederholen*, machte aus einer ansonsten nichtssagenden Anleitung ein ausgeklügeltes Rezept für eine Handlung.

Der wesentliche Gedanke ist hier, dass Sie nicht eine Anweisung nach der anderen befolgen sollten, wenn Sie sich mit Anleitungen

beschäftigen. Stattdessen sollten Sie vielleicht auch einmal zurückgehen. Sie treffen Entscheidungen ("Wenn DAS HAAR TROCKEN IST, dann BENUTZE EINE HAARSPÜLUNG") und Sie wiederholen Schritte ("EINSCHÄUMEN-AUSWASCHEN und dann erneut EINSCHÄUMEN-AUSWASCHEN"). Bei der Anwendungsentwicklung verwenden Sie die ganze Zeit über Entscheidungen und Wiederholungen. Dieses Kapitel behandelt Wiederholungen (Schleifen) in Java.

Anweisungen mehrfach wiederholen (der Java-Befehl »while«)

Hier kommt ein weiteres Ratespiel. Der Computer erzeugt eine Zufallszahl zwischen 1 und 10 und fordert Sie auf, die Zahl zu erraten. Wenn Sie falsch raten, wird das Spiel fortgesetzt. Wenn Sie richtig raten, wird das Spiel beendet. [Listing 6.1](#) zeigt das entsprechende Programm und [Abbildung 6.1](#) eine Spielrunde.

```
import static java.lang.System.out;
import java.util.Scanner;
import java.util.Random;
public class GuessAgain {
    public static void main(String args[]) {
        Scanner keyboard = new Scanner(System.in);
        int numGuesses = 0;
        int randomNumber = new Random().nextInt(10) +
1;

        out.println("          *****          ");
        out.println("Willkommen beim Ratespiel");
        out.println("          *****          ");
        out.println();
        out.print("Geben Sie eine ganze Zahl zwischen
"
                + "1 und 10 ein: ");
        int inputNumber = keyboard.nextInt();
        numGuesses++;
        while (inputNumber != randomNumber) {
            out.println();
```

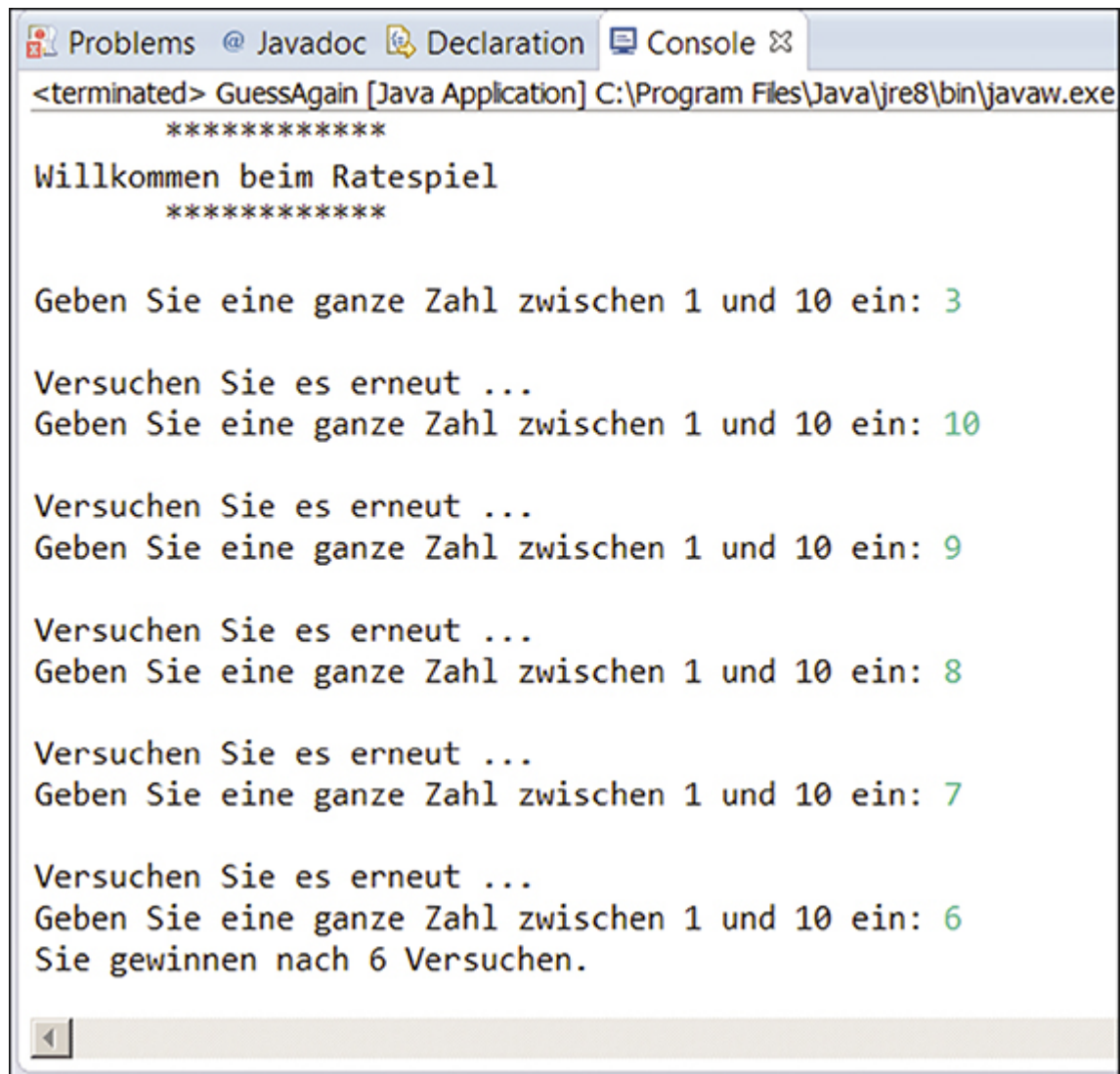
```

        out.println("Versuchen Sie es erneut ...")
    );
    out.print("Geben Sie eine ganze Zahl zwis
chen"
        + " 1 und 10 ein: ");
    inputNumber = keyboard.nextInt();
    numGuesses++;
}
out.print("Sie gewinnen nach ");
out.println(numGuesses + " Versuchen.");
keyboard.close();
}
}

```

Listing 6.1: Ein Ratespiel mit Wiederholungen

In [Abbildung 6.1](#) braucht der Benutzer drei Versuche. Jedes Mal prüft der Computer, ob die geratene Zahl richtig ist. Ist sie falsch, erscheint die Aufforderung, es noch einmal zu versuchen. Bei einer richtigen Antwort wird ein begeistertes »Sie gewinnen« und die Anzahl der Versuche ausgegeben. Der Computer wiederholt mehrere Befehle und prüft bei jedem Durchlauf erneut nach, ob die Zufallszahl richtig geraten wurde. Außerdem fügt er bei jedem dieser Durchläufe seiner Liste mit Rateversuchen 1 hinzu. Wenn der Benutzer richtig geraten hat, zeigt der Computer das Ergebnis dieser Liste an. [Abbildung 6.2](#) gibt die Struktur dieses Ablaufs wieder.

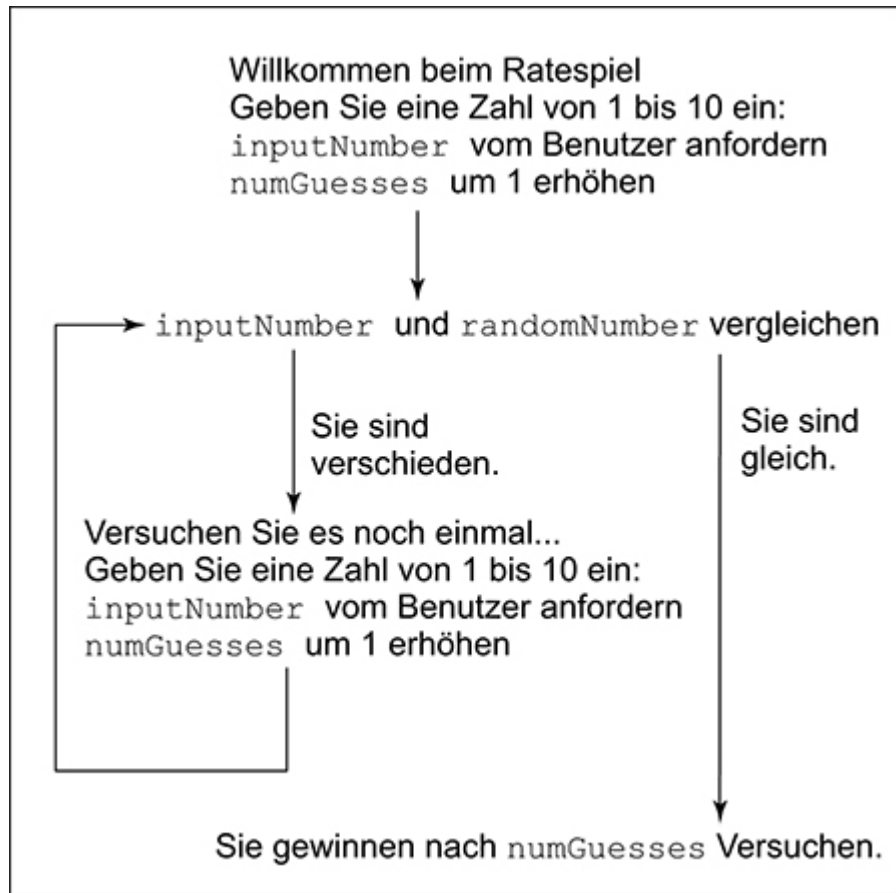


The screenshot shows a Java IDE window with a console tab selected. The console output displays the execution of a Java application named 'GuessAgain'. The program starts with a title 'Willkommen beim Ratespiel' and a series of prompts asking the user to guess a number between 1 and 10. The user's inputs are shown in green text. The program provides feedback after each guess, either prompting for another attempt or announcing a win after six correct guesses.

```
<terminated> GuessAgain [Java Application] C:\Program Files\Java\jre8\bin\javaw.exe
*****
Willkommen beim Ratespiel
*****

Geben Sie eine ganze Zahl zwischen 1 und 10 ein: 3
Versuchen Sie es erneut ...
Geben Sie eine ganze Zahl zwischen 1 und 10 ein: 10
Versuchen Sie es erneut ...
Geben Sie eine ganze Zahl zwischen 1 und 10 ein: 9
Versuchen Sie es erneut ...
Geben Sie eine ganze Zahl zwischen 1 und 10 ein: 8
Versuchen Sie es erneut ...
Geben Sie eine ganze Zahl zwischen 1 und 10 ein: 7
Versuchen Sie es erneut ...
Geben Sie eine ganze Zahl zwischen 1 und 10 ein: 6
Sie gewinnen nach 6 Versuchen.
```

[Abbildung 6.1](#): Spielen Sie, bis Sie umfallen.



[Abbildung 6.2](#): Ein Ratespiel mit Wiederholungen

Wenn Sie noch einmal einen Blick auf [Listing 6.1](#) werfen, sehen Sie den Code, der dies alles erledigt. Im Zentrum steht eine Sache, die `while`-Befehl heißt (und die auch *while-Schleife* genannt wird). In die normale Umgangssprache übersetzt, sagt dieser Befehl:

```
Während (while) inputNumber ungleich randomNumber ist  
,  
führe die Anweisungen in den geschweiften Klammern aus:  
{  
}  
}
```

Die Anweisungen in den geschweiften Klammern (die Anweisungen, die für die Wiederholungen zuständig sind) sorgen dafür, dass Versuchen Sie es erneut ... und Geben Sie eine ganze Zahl zwischen 1 und 10 ein ausgegeben werden, nehmen eine Zahl von der Tastatur an und erhöhen die Anzahl der Versuche um 1.



Wenn Sie es mit Zählern wie `numGuesses` in [Listing 6.1](#) zu tun haben, besteht die Gefahr, sich um 1 nach oben oder unten zu verzählen. Sie können dies vermeiden, indem Sie dafür sorgen, dass die `++`-Befehle und die Anweisungen, deren Ereignisse Sie zählen, dicht nebeneinanderstehen. So fängt zum Beispiel in [Listing 5.1](#) die Variable `numGuess` mit dem Wert 0 an, denn wenn das Programm startet, hat der Benutzer noch keinen Versuch unternommen zu raten. Weiter hinten im Programmverlauf steht dann direkt hinter jedem Aufruf von `keyboard.nextInt` ein `numGuesses++`-Befehl. Auf diese Weise erreichen Sie, dass der Zähler immer dann erhöht wird, wenn der Benutzer einen neuen Rateversuch unternimmt.

Die Befehle in den geschweiften Klammern werden so lange wiederholt, wie `inputNumber != randomNumber` wahr ist. Jede Wiederholung der Befehle in der Schleife wird *Iteration* der Schleife oder *Schleifendurchlauf* genannt. In [Abbildung 6.1](#) durchläuft die Schleife drei Iterationen. Wenn Sie mir nicht glauben, zählen Sie in der Abbildung nach, wie oft `Versuchen Sie es erneut ...` ausgegeben wird. (Diese Aufforderung erscheint bekanntlich nach jedem Fehlversuch beim Raten.)

Wenn Sie lange genug warten, rät der Benutzer irgendwann richtig, und der Computer geht an den Anfang des `while`-Befehls zurück, prüft die Bedingung in den Klammern und findet heraus, dass er eine doppelte Verneinung erreicht hat. Die Ungleich-Beziehung (`!=`) zwischen `inputNumber` und `randomNumber` ist nicht länger wahr. Da dies dazu führt, dass die Bedingung der `while`-Anweisung falsch geworden ist, springt der Computer hinter die `while`-Schleife zu den dort stehenden Befehlen, die dann zum Beispiel Sie gewinnen in 3 Versuchen ausgeben.



So, wie der Code in [Listing 6.1](#) aufgebaut ist, springt der Computer nie aus der Mitte der Schleife heraus. Wenn er herausfindet, dass `inputNumber` ungleich `randomNumber` ist, legt er los und führt die fünf Befehle aus, die sich innerhalb der geschweiften Klammern der Schleife befinden. Der Computer führt diese Prüfung nach einem vollen Durchlauf dieser fünf Befehle erneut aus (um herauszufinden, ob `inputNumber` immer noch ungleich `randomNumber` ist).

Eine bestimmte Anzahl an Wiederholungen festlegen (der Java-Befehl »for«)

»Schreibe hundertmal ›Ich soll während des Unterrichts nicht schwätzen‹ an die Tafel.«

Was Ihr Lehrer damit wirklich gemeint hat, ist:

```
Setze den Zähler auf 0.  
So lange der Zähler kleiner oder gleich 100 ist,  
    Schreibe an die Tafel:  
        ›Ich soll während des Unterrichts nicht schwätze  
n´.  
    Erhöhe den Zähler um 1.
```

Glücklicherweise wussten Sie damals noch nichts von Schleifen und Zählern. Denn wenn Sie davon bei Ihrem Lehrer angefangen hätten, wären Sie in noch größere Schwierigkeiten gekommen.

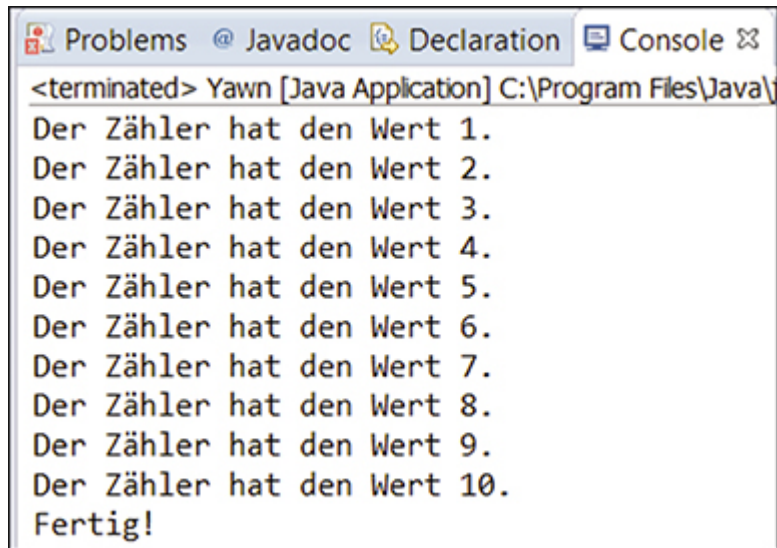
Auf die eine oder die andere Weise stoßen Sie im Leben immer und überall auf zählende Schleifen, und eine gute Programmierung spiegelt das echte Leben wider – oder war das anders herum? Wenn Sie dem Computer mitteilen, was er zu tun hat, gehören hierzu auch die Anweisungen, drei Zeilen auszugeben, zehn Konten zu verarbeiten, eine Million Telefonnummern anzuwählen oder etwas anderes zu erledigen, das ansteht. Da zählende Schleifen zum Alltagsgeschäft der

Programmierung gehören, haben die Entwickler von Programmiersprachen für diese Art von Schleifen besondere Befehle erstellt. In Java heißt der Befehl, der etwas eine bestimmte Anzahl von Wiederholungen durchlaufen lässt, `for`. Die [Listing 6.2](#) und [6.3](#) demonstrieren, wie der `for`-Befehl verwendet wird. Dabei enthält [Listing 6.2](#) ein ganz einfaches und [Listing 6.3](#) ein exotischeres Beispiel. Suchen Sie sich eines aus.

```
import static java.lang.System.out;
public class Yawn {
    public static void main(String args[]) {
        for (int count = 1; count <= 10; count++) {
            out.print("Der Zähler hat den Wert ");
            out.print(count);
            out.println(".");
        }
        out.println("Fertig!");
    }
}
```

[Listing 6.2](#): Die langweiligste `for`-Schleife der Welt

[Abbildung 6.3](#) zeigt Ihnen, was Sie erhalten, wenn Sie das Programm in [Listing 6.2](#) ausführen. Die `for`-Schleife dieses Listings beginnt damit, dass sie den Wert der Variablen `count` auf 1 setzt. Nun prüft der Befehl nach, ob `count` kleiner oder gleich 10 ist (was zu diesem Zeitpunkt sicher ist). Dann legt der `for`-Befehl los und führt die Ausgabebefehle zwischen den geschweiften Klammern aus. (Die erste Anzeige lautet `Der Zähler hat den Wert 1.`) Jetzt erhöht der letzte Befehl in den (runden) Klammern den Wert von `count` um 1.

A screenshot of a Java IDE's console window. The window has a title bar with tabs for 'Problems', 'Javadoc', 'Declaration', and 'Console'. The 'Console' tab is active, showing the output of a Java application named 'Yawn'. The output consists of ten lines, each saying 'Der Zähler hat den Wert' followed by a number from 1 to 10, and a final line saying 'Fertig!'. The path 'C:\Program Files\Java\...' is visible in the title bar.

```
<terminated> Yawn [Java Application] C:\Program Files\Java\Y
Der Zähler hat den Wert 1.
Der Zähler hat den Wert 2.
Der Zähler hat den Wert 3.
Der Zähler hat den Wert 4.
Der Zähler hat den Wert 5.
Der Zähler hat den Wert 6.
Der Zähler hat den Wert 7.
Der Zähler hat den Wert 8.
Der Zähler hat den Wert 9.
Der Zähler hat den Wert 10.
Fertig!
```

[Abbildung 6.3](#): Bis zehn zählen

Die `for`-Anweisung prüft nun, da `count` gleich 2 ist, erneut nach, ob `count` kleiner oder gleich 10 ist. Hier kommt eine Bestätigung zurück (weil 2 bekanntlich kleiner als 10 ist). Die `for`-Schleife kehrt zu den Anweisungen in den geschweiften Klammern zurück und gibt auf dem Bildschirm `Der Zähler hat den Wert 2.` aus. Anschließend wird `count` wieder um 1 erhöht, und das ganze Spiel beginnt von Neuem.

So geht es weiter, bis `count` nach zehn Iterationen den Wert 11 hat. Wenn dies der Fall ist, schlägt der Test, ob `count` kleiner oder gleich 10 ist, fehl, und die Schleife wird beendet. Der Computer springt nun zu dem Befehl, der unmittelbar auf die `for`-Schleife folgt. In [Listing 6.2](#) gibt der Computer `Fertig!` aus.

[Abbildung 6.4](#) stellt den Ablauf noch einmal grafisch dar.

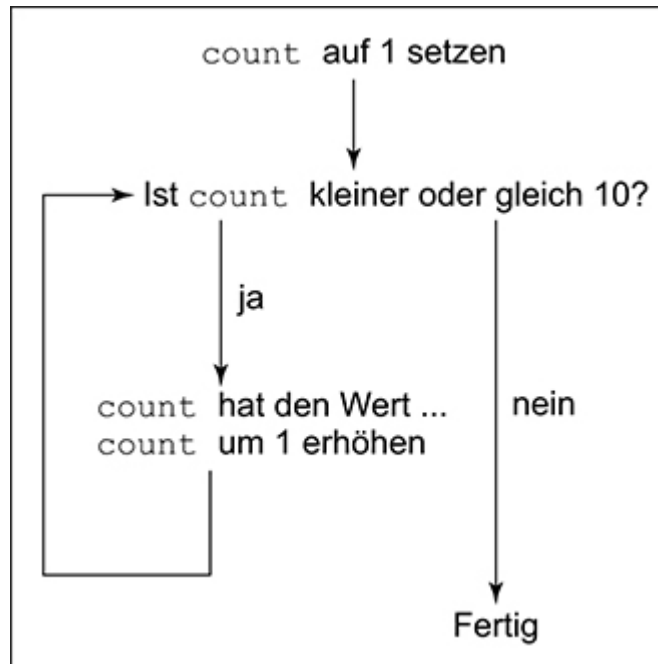


Abbildung 6.4: Die Struktur der *for*-Schleife in Listing 6.2

Die Anatomie eines »for«-Befehls

Auf das Schlüsselwort `for` folgt eine Klammer, die immer drei Dinge enthält: eine *Initialisierung*, einen *Ausdruck* und eine *Aktualisierung*.

Jedes dieser drei Elemente in den Klammern erfüllt einen ganz bestimmten Zweck:

- ✓ Die *Initialisierung* wird ausgeführt, wenn das Programm den `for`-Befehl zum ersten Mal erreicht.
- ✓ Der *Ausdruck* wird mehrfach (vor jeder Iteration) ausgewertet.
- ✓ Auch die *Aktualisierung* wird mehrfach (am Ende jeder Iteration) vorgenommen.

Vielleicht hilft es, wenn Sie sich die Schleife mit umgebautem Text vor Augen führen:

```
int count = 1
for count <= 10 {
    out.print("Der Zähler hat den Wert ");
    out.print(count);
    out.println(".");
}
```

```
        count++  
    }
```

Sie dürfen auf diese Weise keinen echten `for`-Befehl schreiben. Hier geht es nur darum, Ihnen zu zeigen, in welcher Reihenfolge die Teile des Befehls ausgeführt werden.



Wenn Sie in der Initialisierung einer `for`-Schleife eine Variable deklarieren, kann diese nur innerhalb der Schleife verwendet werden. So erhalten Sie zum Beispiel eine Fehlermeldung, wenn Sie in [Listing 6.2](#) `out.println(count)` nach dem Ende der Schleife verwenden.



Sie können alles, was Sie mit einer `for`-Schleife machen, auch mit einer `while`-Schleife durchführen. Ob Sie sich für eine `for`-Schleife entscheiden, ist eine Frage des Stils und der Bequemlichkeit, und nicht eine der anstehenden Aufgaben.

Die Weltpremiere von »Al's All Wet«

[Listing 6.2](#) ist zwar ganz nett, aber das Programm in diesem Listing ist ziemlich langweilig. Wenn Sie etwas Interessanteres sehen wollen, sollten Sie sich mit [Listing 6.4](#) beschäftigen. Es erfüllt das Versprechen, das ich in [Kapitel 5](#) gebe, und zeigt den gesamten Text des Hits *Al's All Wet* auf dem Bildschirm an.

```
import static java.lang.System.out;  
public class AlsAllWet {  
    public static void main(String args[]) {  
        for (int verse = 1; verse <= 3; verse++) {  
            out.print("Al's all wet. ");  
            out.println("Oh, why is Al all wet? Oh,");  
            out.print("Al's all wet 'cause ");  
            out.println("he's standing in the rain.");  
            out.println("Why is Al out in the rain?");  
            switch (verse) {  
                case 1:  
                    out.println("That  
's because he has no brain.");  
            }  
        }  
    }  
}
```

```

        break;
    case 2:
        out.println("That
's because he is a pain.");
        break;
    case 3:
        out.println("'Cause this is the last refrai
n.");
        break;
    }
    switch (verse) {
    case 3:
        out.println("Last refrain, last refrain,");
    case 2:
        out.println("He's a pain, he's a pain,");
    case 1:
        out.println("Has no brain, has no brain,");
    }
    out.println("In the rain, in the rain.");
    out.println("Ohhhhhhhh...");
    out.println();
}
out.print("Al's all wet. ");
out.println("Oh, why is Al all wet? Oh,");
out.print("Al's all wet 'cause ");
out.println("he's standing in the rain.");
}
}

```

Listing 6.3: Der vollständige Text von »Al's All Wet«

Listing 6.3 kombiniert viele Ideen aus den [Kapiteln 5](#) und 6. In diesem Listing sind zwei `switch`-Befehle in einer `for`-Schleife verschachtelt. Einer dieser `switch`-Befehle verwendet `break`-Anweisungen, während der andere das Programm durchfallen lässt. Da der Wert der Zählervariablen der Schleife (`verse`) die Werte 1, 2 und 3 annimmt, werden alle `case`-Klauseln im `switch`-Befehl ausgeführt. Wenn das Programm dann fast fertig ist und die `for`-Schleife verlassen hat, sorgen die restlichen vier Befehle dafür, dass die letzte Strophe des Songs ausgegeben wird.



Wenn ich lautstark verkündet habe, dass ein `for`-Befehl primär zum Zählen verwendet wird, gehe ich großzügig mit der Wahrheit um. Javas `for`-Befehl ist sehr vielseitig einsetzbar. Sie können einen `for`-Befehl in Situationen einsetzen, die nichts mit Zählen zu tun haben. So läuft zum Beispiel eine `for`-Anweisung, die wie `for( i = 0; i < 10; )` keinen Aktualisierungsteil enthält, endlos weiter. Die Schleife wird beendet, wenn eine Aktion in der Schleife der Variablen `i` einen großen Wert zuweist. Sie können sogar einen `for`-Befehl erstellen, dessen Klammerpaar leer ist. Die Schleife `for ( ; ; )` läuft endlos, was sinnvoll ist, wenn sie zum Beispiel eine Maschine steuert. Normalerweise achten Sie beim Schreiben einer `for`-Anweisung aber doch darauf, wie oft Befehle wiederholt werden sollen.



[Listing 6.3](#) verwendet `break`-Befehle, um aus einer `switch`-Anweisung auszusteigen. Ein `break` kann aber auch in einer Schleife eine Rolle spielen, wie ein Beispiel auf der amerikanischen Website dieses Buches unter www.allmycode.com/JavaForDummies zeigt.

Wiederholen, bis Ihr Wunsch erfüllt wird (der Java-Befehl »do«)

Heute möchte ich jung und albern sein (auf jeden Fall albern). Schauen Sie sich noch einmal [Abbildung 6.2](#) an und achten Sie darauf, wie die `while`-Schleife funktioniert. Wenn die Programmausführung in eine `while`-Schleife einsteigt, führt der Computer eine Prüfung durch, um sicherzustellen, dass die Bedingung der Schleife wahr ist. Ist dies nicht der Fall, werden die Anweisungen innerhalb der Schleife nicht ausgeführt. Sie könnten

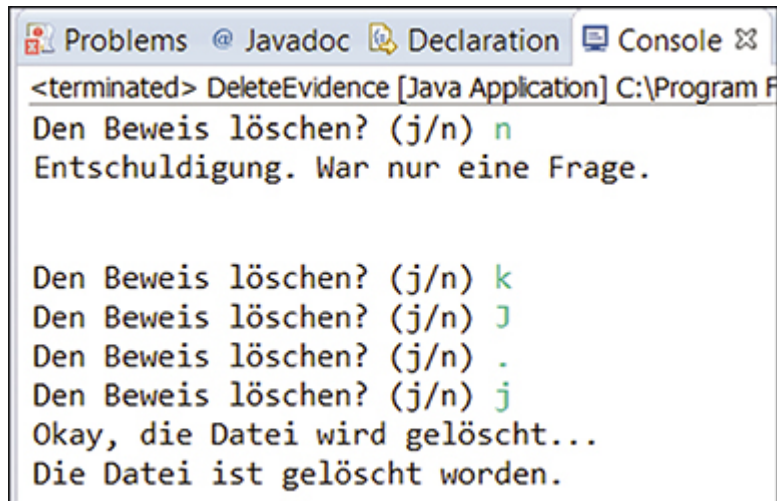
problemlos eine `while`-Schleife herstellen, deren Befehle niemals ausgeführt werden (obwohl ich mir keinen Grund für eine solche Programmierung vorstellen kann).

```
int twoPlusTwo = 2 + 2;
while (twoPlusTwo == 5) {
    out.println("Machen Sie Witze?");
    out.println("2 + 2 ist nicht gleich 5");
    out.print("Das weiß jedes Kind.");
    out.println(" 2 + 2 ist gleich 3");
}
```

Trotz dieses albernem `twoPlusTwo`-Beispiels hat es sich gezeigt, dass der `while`-Befehl das am vielseitigsten einsetzbare Schleifenkonstrukt Javas ist. Die `while`-Schleife hat sich insbesondere in Situationen bewährt, in denen Sie schauen müssen, bevor Sie loslegen. Zum Beispiel: »Während (`while`) Geld auf meinem Konto ist, bezahle die monatliche Kreditrate.« Hier gilt also, dass die Überweisung nur erfolgen soll, wenn Sie auf Ihrem Konto Geld haben.

Gelegentlich kommt es aber auch vor, dass Sie »springen«, bevor Sie schauen, wohin es geht. Nehmen Sie zum Beispiel die Situation, in der Sie den Benutzer auffordern, zu antworten. Es kann nun sein, dass die Eingabe des Benutzers sinnvoll ist, aber auch das Gegenteil ist denkbar. In diesem zweiten Fall möchten Sie die Aufforderung vielleicht wiederholen. Es könnte ja sein, dass der Benutzer nur eine falsche Taste erwischt oder die Frage nicht verstanden hat.

[Abbildung 6.5](#) zeigt einige Ausführungen eines Programms, das eine Datei löschen soll. Das Programm fragt den Benutzer, ob es in Ordnung ist, wenn es die Datei löscht. Wenn der Benutzer `j` oder `n` eingibt, macht das Programm entsprechend seiner Eingabe weiter. Wenn der Benutzer aber einen anderen Buchstaben eingibt, wird er vom Programm aufgefordert, seine Antwort noch einmal einzugeben.



```
<terminated> DeleteEvidence [Java Application] C:\Program F
Den Beweis löschen? (j/n) n
Entschuldigung. War nur eine Frage.

Den Beweis löschen? (j/n) k
Den Beweis löschen? (j/n) J
Den Beweis löschen? (j/n) .
Den Beweis löschen? (j/n) j
Okay, die Datei wird gelöscht...
Die Datei ist gelöscht worden.
```

Abbildung 6.5: Der Code in [Listing 6.4](#) wird zweimal ausgeführt.

Um dieses Programm zu schreiben, benötigen Sie eine Schleife – eine Schleife, die den Benutzer wiederholt fragt, ob eine Datei gelöscht werden soll. Die Schleife fragt so lange, bis der Benutzer eine eindeutige Antwort gibt. Das Bemerkenswerte an dieser Schleife ist, dass sie nichts überprüfen muss, bevor der Benutzer nicht ein erstes Mal gefragt worden ist – vorher bekommt die Schleife nichts zu tun. Sie startet also nicht mit »so lange dies und das wahr ist, erhalte vom Benutzer eine Antwort«. Stattdessen legt die Schleife los, erhält vom Benutzer eine Antwort und überprüft dann diese Antwort, um herauszufinden, ob sie brauchbar ist.

Das ist der Grund dafür, dass das Programm in [Listing 6.4](#) eine `do`-Schleife enthält (die auch *do ... while-Schleife* genannt wird). Bei einer `do`-Schleife springt das Programm direkt in die Schleife, führt die Anweisungen dort aus und überprüft dann erst eine Bedingung, um herauszufinden, ob das Ergebnis der ausgeführten Aktion sinnvoll ist oder nicht. Wenn das Ergebnis der Antwort positiv verarbeitet werden kann, wird die Schleife beendet. Anderenfalls begibt sich das Programm wieder an den Anfang der Schleife, um einen neuen Durchlauf zu starten.

```
import java.io.File;
import static java.lang.System.out;
import java.util.Scanner;
```

```

public class DeleteEvidence {
    public static void main(String args[]) {
        File evidence = new File("frisierteKonten.txt");
;
        Scanner keyboard = new Scanner(System.in);
        char reply;
        do {
            out.print("Den Beweis löschen? (j/n) ");
            reply =
                keyboard.findWithinHorizon(".",0).charAt(
0);
            } while (reply != 'j' && reply != 'n');
            if (reply == 'j') {
                out.println("Okay, die Datei wird gelöscht..
.");
                evidence.delete();
                out.println("Die Datei ist gelöscht worden."
);
            } else {
                out.println("Entschuldigung, war nur eine "
+ "Frage.");
            }
            keyboard.close();
        }
    }
}

```

[Listing 6.4](#): Löschen oder nicht löschen, das ist hier die Frage.

[Abbildung 6.5](#) zeigt zwei Ausführungen des Codes in [Listing 6.4](#). Das Programm akzeptiert nur die Kleinbuchstaben j und n. Die Großbuchstaben J und N sind keine gültigen Antworten. Damit das Programm aber auch mit solchen Antworten klarkommt, ändern Sie den Code wie folgt:

```

do {
    out.print("Delete evidence? (y/n) ");
    reply = keyboard.findWithinHorizon(".", 0).charAt
(0);
} while (reply != 'y' && reply != 'Y' &&
        reply != 'n' && reply != 'N');

if (reply == 'y' || reply == 'Y') {

```

[Abbildung 6.6](#) zeigt den Programmablauf in der Schleife von [Listing 6.4](#). Bei einer do-Schleife kann es nie zu einer Situation

wie der im Programm `twoPlusTwo` kommen, das am Anfang dieses Abschnitts gezeigt wird. Da eine `do`-Schleife ihre erste Aktion ausführt, ohne eine Bedingung zu testen, wird sie immer mindestens einmal durchlaufen.

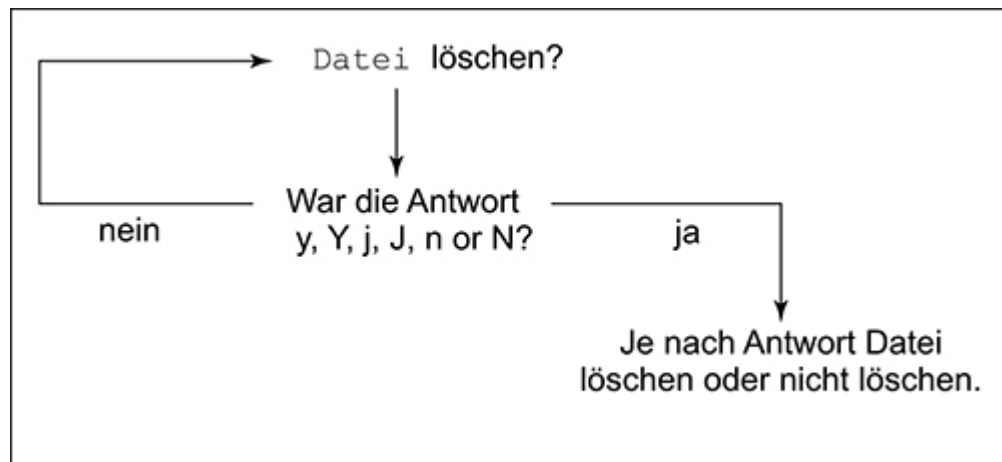


Abbildung 6.6: Der Ablauf einer `do`-Schleife



Der Ort, an dem sich die in [Listing 6.4](#) erwähnte Datei `frisierteKonten.txt` auf Ihrer Festplatte befindet, hängt von verschiedenen Dingen ab. Wenn Sie eine solche Datei im falschen Verzeichnis erstellen, kann der Code sie nicht löschen. In den meisten Fällen beginnen Sie die Tests von [Listing 6.4](#) damit, dass Sie in Ihrer IDE ein neues Projekt anlegen. Dieses Projekt »lebt« in einem Ordner auf Ihrer Festplatte, und die Datei `frisierteKonten.txt` befindet sich in diesem Ordner. Ich arbeite mit der Eclipse-IDE, und dort gibt es ein Projekt mit dem Namen `Listing06-04`. Dieses Projekt befindet sich auf meiner Festplatte in einem Ordner mit dem Namen

`C:\Benutzermein_Benutzernameworkspace\Listing06-04`. Dort gibt es die Datei `frisierteKonten.txt` (die ich mit dem Code in Listing 6.4 lösche). Außerdem gibt es in dem Ordner `Listing06-04` einen Unterordner mit dem Namen `src`. Und genau in diesem Ordner befindet sich

meine Datei `DeleteEvidence.java`, die den Code von [Listing 6.4](#) enthält. (Sie erfahren in [Kapitel 8](#) mehr über Dateien und ihre Ordner.)

Ein einzelnes Zeichen lesen

Gehen wir weit zurück – bis zu [Listing 5.3](#). Dort gibt der Benutzer ein Wort über die Tastatur ein. Die Methode `keyboard.next` schnappt sich das Wort und bringt es in einer `String`-Variablen mit dem Namen `password` unter. Das klappt deswegen so gut, weil eine `String`-Variable viele Zeichen auf einmal speichern kann und die Methode `next` in der Lage ist, viele Zeichen auf einmal auszulesen.

In [Listing 6.4](#) sind Sie nicht daran interessiert, mehrere Zeichen auszulesen. Vom Benutzer wird erwartet, dass er nur einen Buchstaben eingibt – entweder `j` oder `n`. Deshalb müssen Sie keine `String`-Variable anlegen, um die Rückmeldung des Benutzers zu speichern. Stattdessen erstellen Sie eine `char`-Variable – eine Variable, die immer nur ein Zeichen aufnehmen kann.

Die Java-API kennt keine Methode `nextChar`. Deshalb müssen Sie improvisieren, wenn es darum geht, etwas auszulesen, das in einer `char`-Variablen gespeichert werden kann.

Dieses Improvisieren sieht in [Listing 6.4](#) so aus:

```
keyboard.findWithinHorizon(".", 0).charAt(0)
```

Wenn Sie einmal ein einzelnes Zeichen auslesen wollen, können Sie diesen Code nahtlos aus [Listing 6.4](#) übernehmen.



Eine `String`-Variable kann viele oder nur ein Zeichen enthalten. Eine `String`-Variable, die nur ein Zeichen enthält, wird deswegen aber nicht zu einer `char`-Variablen. `String`- und `char`-Variablen werden unabhängig davon, was Sie in ihnen ablegen, unterschiedlich behandelt.

Java und die Behandlung von Dateien

Wir müssen uns in [Listing 6.4](#) die Befehle zur Dateibehandlung ein wenig näher anschauen. Zu diesen Befehlen gehören Klassen, Objekte und Methoden. Viele Einzelheiten zu diesen Dingen stehen zwar in anderen Kapiteln wie den [Kapiteln 7](#) und 9, aber ich mache nichts falsch, wenn ich schon an dieser Stelle auf ein paar Besonderheiten hinweise.

In der Sprach-API von Java gibt es eine Klasse mit dem Namen `java.io.File`. Der Befehl

```
File evidence = new File("frisierteKonten.txt");
```

erstellt im Arbeitsspeicher des Computers ein neues Objekt. Dieses Objekt, das aus der Klasse `java.io.File` gebildet wird, beschreibt alles, was das Programm über die Datei `frisierteKonten.txt` wissen muss. Von diesem Augenblick an verweist in [Listing 6.4](#) die Variable `evidence` auf die Datei `frisierteKonten.txt`.

Das Objekt `evidence`, das eine Instanz der Klasse `java.io.File` ist, verfügt über die Methode `delete`. (Was ich dazu sagen kann? Es steht alles in der API-Dokumentation.) Wenn Sie `evidence.delete` aufrufen, sorgt der Computer dafür, dass die Datei verschwindet.

Es sollte klar sein, dass Sie nichts loswerden können, was nicht existiert. Wenn Sie den Konstruktor

```
File evidence = new File("frisierteKonten.txt");
```

aufrufen, prüft Java nicht nach, ob eine solche Datei wirklich vorhanden ist. Um Java zu einer solchen Prüfung zu zwingen, stehen Ihnen ein paar Möglichkeiten zur Verfügung. Am einfachsten ist es, die Methode `exists` aufzurufen. Wenn Sie `evidence.exists()` verwenden, schaut Java in dem Ordner nach, in dem es die Datei `frisierteKonten.txt` vermutet. Der Aufruf gibt `true` zurück, wenn Java die Datei in diesem Ordner findet. Ansonsten wird `false` zurückgegeben.

Hier eine überarbeitete Version von [Listing 6.4](#), bei der es im Code einen Aufruf von `exists` gibt:

```
import java.io.File;
import static java.lang.System.out;
import java.util.Scanner;
public class DeleteEvidence {
    public static void main(String args[]) {
        File evidence = new File("frisierteKonten.txt");
        if (evidence.exists()) {
            Scanner keyboard = new Scanner(System.in);
            char reply;
            do {
                out.print("Den Beweis löschen? (j/n) ");
                reply =
                    keyboard.findWithinHorizon(".", 0).charAt(
0);
                } while (reply != 'j' && reply != 'n');
                if (reply == 'y') {
                    out.println("Okay, die Datei wird gelöscht...
");
                    evidence.delete();
                    out.println("Die Datei ist gelöscht worden.");
                }
                } else {
                    out.println("Entschuldigung. War nur eine "
+ "Frage.");
                }
                keyboard.close();
            }
        }
    }
```

Variablendeklarationen und Blöcke

Befehle, die von geschweiften Klammern umgeben sind, bilden einen Block. Wenn Sie eine Variable in einem Block deklarieren, können Sie sie nicht außerhalb dieses Blockes verwenden. So würden Sie zum Beispiel in [Listing 6.4](#) eine Fehlermeldung erhalten, wenn Sie folgende Änderungen vornähmen:

```
do {
    out.print("Delete evidence? (j/n) ");
    char reply =
        keyboard.findWithinHorizon(".", 0).charAt(0);
```



```
} while (reply != 'j' && reply != 'n');  
if (reply == 'j')
```

Durch die Deklaration von `char reply` innerhalb der geschweiften Klammern der Schleife ist die Verwendung von `reply` außerhalb der Klammern nicht mehr zulässig. Wenn Sie versuchen, diesen Code zu kompilieren, erhalten Sie drei Fehlermeldungen – zwei für `reply` in `while (reply != 'j' && reply != 'n')` und eine dritte für `reply` im `if`-Befehl.

Damit sind Ihnen in [Listing 6.4](#) die Hände gebunden. Das Programm verwendet die Variable `reply` zum ersten Mal in der Schleife. Wenn Sie aber auch noch nach der Schleife auf diese Variable zugreifen wollen, müssen Sie `reply` vor der Schleife deklarieren. Und dann fahren Sie am besten damit, diese Deklaration ohne Initialisierung vorzunehmen. ([Kapitel 4](#) enthält weitere Informationen über das Deklarieren von Variablen, und Blöcke behandle ich ausführlicher in [Kapitel 5](#).)

Die drei Schleifenarten, die in diesem Kapitel beschrieben werden (`while`-Schleifen, `for`-Schleifen und `do ... while`-Schleifen), gibt es in allen Java-Versionen. Zusätzlich kennen neuere Java-Versionen (ab Java 5) eine Schleifenart, die *erweiterte for-Schleife* heißt. Ich gehe in [Kapitel 11](#) näher auf diesen Schleifentyp ein.

Teil III

Das große Ganze im Auge behalten: Die objektorientierte Programmierung



In diesem Teil . . .

- ✓ Sie finden heraus, was Klassen und Objekte sind.
- ✓ Sie finden heraus, wie die objektorientierte Programmierung hilft, vorhandenen Code wiederzuverwenden (was Zeit und Geld spart).
- ✓ Sie sind der Beherrscher Ihrer eigenen virtuellen Welt, indem Sie ganz neue Objekte entwerfen.

7

In Begriffen wie Klassen und Objekte denken

In diesem Kapitel

- ▶ Wie ein richtiger objektorientierter Programmierer denken
 - ▶ Werte an Methoden übergeben und von dort abholen
 - ▶ Die kleinen Geheimnisse im objektorientierten Code ausblenden
-

Als Autor von Computerbüchern ist mir immer wieder gesagt worden, dass ich nicht davon ausgehen darf, dass Abschnitte und Kapitel in ihrer logischen Reihenfolge gelesen werden. Im Buch wird geblättert und das herausgepickt, was gerade benötigt wird; die Dinge, von denen man glaubt, dass sie überflüssig sind, werden übersprungen. Daran dachte ich, als mir der Gedanke kam, dass Sie [Kapitel 1](#) übersprungen haben könnten. Sollte dies der Fall sein, müssen Sie keine Schuldgefühle haben. Sie können alles in 60 Sekunden wiedergutmachen, indem Sie diese komprimierte Zusammenfassung von [Kapitel 1](#) lesen:

Da Java eine objektorientierte Programmiersprache ist, ist es ihr primäres Ziel, Klassen und Objekte zu beschreiben. Eine Klasse entspricht dem Begriff eines Gegenstandes. Ein Objekt ist eine konkrete Instanz einer Klasse. Der Programmierer definiert eine Klasse, und der Computer erstellt aus der Klassendefinition die einzelnen Objekte.

Natürlich können Sie diese Zusammenfassung ignorieren, weil sie Ihnen immer noch zu lang ist. In diesem Fall lässt sich [Kapitel 1](#) noch weiter kürzen:

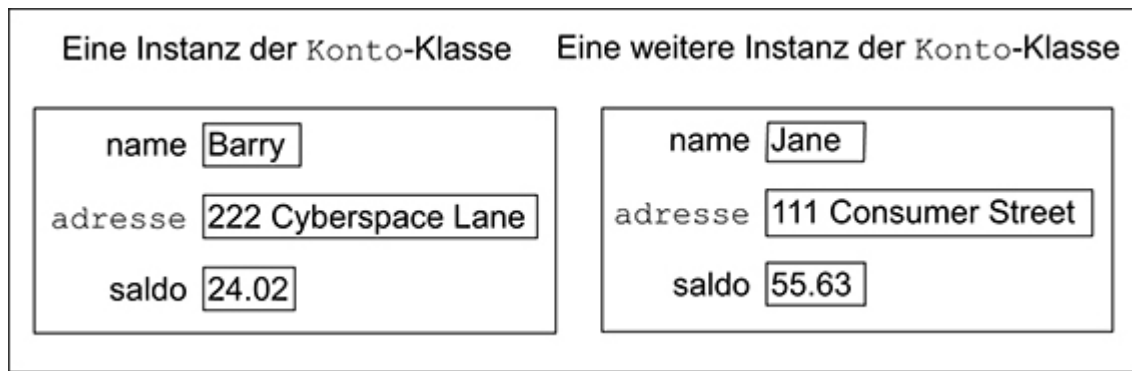
Klassen; Objekte

Eine Klasse definieren (was es heißt, ein Konto zu sein)

Was unterscheidet ein Konto von einem anderen Konto? Wenn Sie einen Mitarbeiter einer Bank fragen, werden Sie in ein längeres Verkaufsgespräch verwickelt. Sie hören dann von Zinssätzen, Gebühren, Überziehungszinsen und vielem mehr. Sie haben Glück, dass ich daran nicht interessiert bin. Ich möchte nur wissen, wodurch sich mein Konto von Ihrem unterscheidet. Schließlich heißt mein Konto *Barry Burd, Burd Brain Consulting*, während Ihres vielleicht *Jane Q. Public, Angehende Java-Expertin* heißt. Mein Kontostand beträgt 24,02 Euro. Und Ihrer?

Letztendlich können die Unterschiede zwischen Konten als Werte von Variablen beschrieben werden. Nehmen wir an, es gibt eine Variable mit dem Namen `saldo`. Bei meinem Konto hat `saldo` den Wert 24,02, während er bei Ihrem Konto den Wert 55,63 hat. Wenn ich nun ein Computerprogramm schreibe, das mit der Verwaltung von Konten zu tun hat, stehe ich vor der Frage, wie ich meine Variable `saldo` von Ihrer Variablen `saldo` trenne.

Die Antwort lautet: Es werden zwei separate Objekte erstellt, die jeweils eine der `saldo`-Variablen enthalten. Und wenn Sie einmal dabei sind, fügen Sie jedem der Objekte noch eine Variable `name` und eine Variable `adresse` hinzu. Damit sind Sie ein ganzes Stück weiter – zwei Objekte, die jeweils ein Konto darstellen. Oder genauer: zwei Objekte, die jeweils eine Instanz der Klasse `Konto` sind (siehe [Abbildung 7.1](#)).



[Abbildung 7.1](#): Zwei Objekte

So weit, so gut, aber das ursprüngliche Problem haben Sie damit leider nicht gelöst. Wie verweisen Sie nun in einem Programm statt auf Ihre Variable `saldo` auf meine? (Für *verweisen* lesen Sie gerne auch *referenzieren*, wobei es dann aber nicht *referenzieren auf etwas*, sondern *etwas referenzieren* heißt.) Glücklicherweise haben Sie zwei Objekte, auf die Sie viel-leicht über Variablen verweisen könnten. Erstellen Sie zwei Variablen – eine mit dem Namen `meinKonto` und eine mit dem Namen `ihrKonto`. Die Variable `meinKonto` verweist auf mein Objekt (meine Instanz der Klasse `Konto`) und seinen gesamten Inhalt. Um dann auf meinen Saldo zu verweisen, schreiben Sie

```
meinKonto.saldo
```

Um auf meinen Namen zu verweisen, schreiben Sie

```
meinKonto.name
```

`ihrKonto.saldo` verweist auf den Wert in der Variablen `saldo` Ihres Objekts und `ihrKonto.name` verweist auf den Wert in der Variablen `name` Ihres Objekts. Um dem Computer mitzuteilen, wie viel Geld ich auf meinem Konto habe, schreiben Sie

```
meinKonto.saldo = 24.02;
```

Wenn Sie Ihren Namen auf dem Bildschirm ausgeben wollen, können Sie Folgendes schreiben:

```
out.println(ihrKonto.name);
```

[Listing 7.1](#) und 7.2 fassen obige Gedankengänge zusammen.

```
public class Konto {  
    String name;  
    String adresse;
```

```
double saldo;
```

[Listing 7.1](#): Das macht ein Konto aus.

Die Klasse `Konto` definiert in [Listing 7.1](#) das, was ein Konto ausmacht. Dieses Listing weist Sie darauf hin, dass jede Instanz der Klasse `Konto` über drei Variablen verfügt – `name`, `adresse` und `saldo`. Dies stimmt mit den Informationen in [Abbildung 7.1](#) überein. Java-Programmierer haben für Variablen dieser Art (für Variablen, die zu den Instanzen einer Klasse gehören) einen besonderen Namen. Jede dieser Variablen – `name`, `adresse` und `saldo` – wird *Feld* genannt.



Eine Variable, die zwar innerhalb einer Klasse, nicht aber innerhalb einer Methode definiert wird, ist ein *Feld*. In [Listing 7.1](#) sind die Variablen `name`, `adresse` und `saldo` Felder. Ein anderer Name für ein Feld ist *Instanzenvariable*.

Wenn Sie sich mit dem Material der [Kapitel 4](#) bis 6 beschäftigt haben, sind Sie vielleicht über den Code für die Klasse `Konto` (in [Listing 7.1](#)) mehr als überrascht. Lässt sich eine komplette Java-Klasse in nur vier Codezeilen (gegebenenfalls zuzüglich separater Zeilen für die geschweiften Klammern) definieren? Ja, das geht. Die Klasse `Konto` in [Listing 7.1](#) ist ein ziemlich gutes Abbild dessen, was Java-Programmierer unter einer Klasse verstehen: Eine Klasse ist eine Gruppe von Elementen. In der Klasse `Konto` von [Listing 7.1](#) sind dies zwei `String`-Werte und ein `double`-Wert.

Variablen deklarieren und Objekte erstellen

Während ich die Straße entlanggehe, nähert sich mir ein junger Mann. Er fordert mich auf, *Sie werden Java lieben!* zu schreiben, und ich mache das. Ich gebe den Text mit Kreide auf dem Bürgersteig aus. Aber das ist eigentlich uninteressant. Wichtig ist nur, dass jemand Anweisungen gibt, denen ich folge.

Später am Tag sitzt eine ältere Frau neben mir auf einer Parkbank. Sie sagt: »Zu einem Konto gehören ein Name, eine Adresse und ein Saldo.« Und ich antworte: »Das stimmt, aber was möchten Sie, dass ich tue?« Ihre Reaktion sieht nun so aus, dass sie mich nur anstarrt und ich deshalb ihre Äußerungen zum Konto einfach Äußerungen sein lasse. Ich sitze, sie sitzt, und wir beide machen absolut nichts.

[Listing 7.1](#) ist wie die ältere Frau. Dieses Listing definiert das, was ein `Konto` ausmacht, aber es sagt mir einfach nicht, was ich mit meinem Konto oder dem Konto einer anderen Person machen soll. Damit etwas geschieht, benötige ich zusätzlichen Code. Ich brauche eine zweite Klasse – eine Klasse, die eine `main`-Methode enthält. Während die ältere Frau und ich ruhig auf der Parkbank sitzen, kommt glücklicherweise ein Kind mit [Listing 7.2](#) vorbei.

```
import static java.lang.System.out;
public class NutzKonto {
    public static void main(String args[]) {
        Konto meinKonto;
        Konto ihrKonto;
        meinKonto = new Konto();
        ihrKonto = new Konto();
        meinKonto.name = "Barry Burd";
        meinKonto.adresse = "222 Cyberspace Lane";
        meinKonto.saldo = 24.02;
        ihrKonto.name = "Jane Q. Public";
        ihrKonto.adresse = "111 Consumer Street";
        ihrKonto.saldo = 55.63;
        out.print(meinKonto.name);
        out.print(" ");
        out.print(meinKonto.adresse);
        out.print(" besitzt € ");
        out.print(meinKonto.saldo);
        out.println();
        out.print(ihrKonto.name);
        out.print(" ");
        out.print(ihrKonto.adresse);
        out.print(" besitzt € ");
        out.print(ihrKonto.saldo);
    }
}
```

[Listing 7.2](#): Mit *Konto*-Objekten arbeiten

Die beiden Klassen – `Konto` und `NutzKonto` – bilden zusammen ein vollständiges Programm. Der Code in [Listing 7.2](#) definiert die Klasse `NutzKonto`, die über eine Methode `main` verfügt. Diese Methode enthält eigene Variablen – `ihrKonto` und `meinKonto`.

Die ersten beiden Zeilen der Methode `main` führen in [Listing 7.2](#) in die Irre. Es kann vorkommen, dass `Konto ihrKonto` so verstanden wird, als wenn es »`ihrKonto` ist ein `Konto`« oder »Die Variable `ihrKonto` verweist auf eine Instanz der Klasse `Konto`« bedeute. Das ist aber nicht das, was die erste Zeile aussagt. Stattdessen meint `Konto ihrKonto`: »Falls und wenn ich die Variable `ihrKonto` auf etwas verweisen lasse, wird dieses Etwas eine Instanz der Klasse `Konto` sein.« Worin besteht der Unterschied?

Der Unterschied besteht darin, dass die einfache Deklaration `Konto ihrKonto` nicht ausreicht, um die Variable `ihrKonto` auf ein Objekt verweisen zu lassen (das Objekt *zu referenzieren*). Die Deklaration reserviert lediglich den Variablennamen `ihrKonto`, damit er später verwendet werden kann, um auf eine Instanz der Klasse `Konto` zu verweisen. Das eigentliche Objekt wird erst später im Code erstellt, wenn der Computer `new Konto()` ausführt.



Aus technischer Sicht sieht es so aus, dass Sie, wenn der Computer `new Konto()` ausführt, ein Objekt erstellen, indem Sie den *Konstruktor* der Klasse `Konto` aufrufen.

Wenn der Computer die Zuweisung `ihrKonto = new Konto()` ausführt, erstellt er ein neues Objekt (eine neue Instanz der Klasse `Konto`) und lässt die Variable `ihrKonto` auf dieses neue Objekt verweisen. (Dafür sorgt das Gleichheitszeichen.)

[Abbildung 7.3](#) verdeutlicht diese Situation.

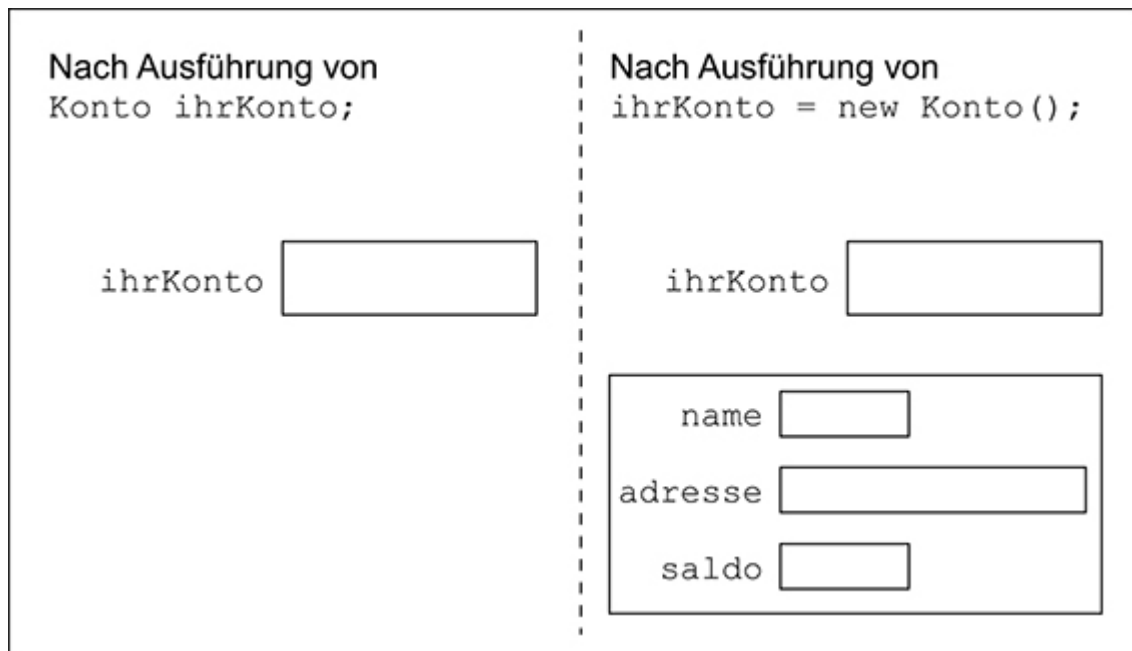


Abbildung 7.2: Vor und nach dem Aufruf eines Konstruktors

Um die Behauptungen zu testen, die ich in den letzten Absätzen aufstelle, habe ich den Code in [Listing 7.2](#) um eine Zeile erweitert. Ich habe versucht, `ihrKonto.name` auszugeben, nachdem ich `ihrKonto` zwar deklariert, aber `new Konto()` noch nicht aufgerufen habe.

```
Konto meinKonto;  
Konto ihrKonto;  
  
out.println(ihrKonto.name);  
meinKonto = new Konto();  
ihrKonto = new Konto();
```

Als ich versucht habe, den neuen Code zu kompilieren, erhielt ich diese Fehlermeldung:

Variable `ihrKonto` might not have been initialized. (Diese Meldung besagt, dass die Variable `ihrKonto` möglicherweise nicht initialisiert worden ist.) Damit sollten alle Fragen beseitigt sein. Bevor Sie nicht `new Konto()` ausführen, ist es unmöglich, die Variable `name` eines Objekts auszugeben, weil dieses Objekt noch gar nicht existiert.



Wenn eine Variable einen Referenztyp hat, reicht es nicht aus, die Variable einfach nur zu deklarieren. Sie erhalten erst dann ein Objekt, wenn Sie einen Konstruktor aufrufen und das Schlüsselwort `new` verwenden. (Ich gehe in [Kapitel 4](#) näher auf Referenztypen ein.)

Eine Variable initialisieren

Ich behaupte in [Kapitel 4](#), dass Sie eine primitive Variable schon bei ihrer Deklaration initialisieren können.

```
int weightOfAPerson = 150;
```

Sie können dasselbe auch mit Referenztyp-Variablen wie `meinKonto` oder `ihrKonto` in [Listing 7.2](#) machen. Die ersten vier Zeilen der Methode `main` des Listings lassen sich in zwei Zeilen zusammenfassen:

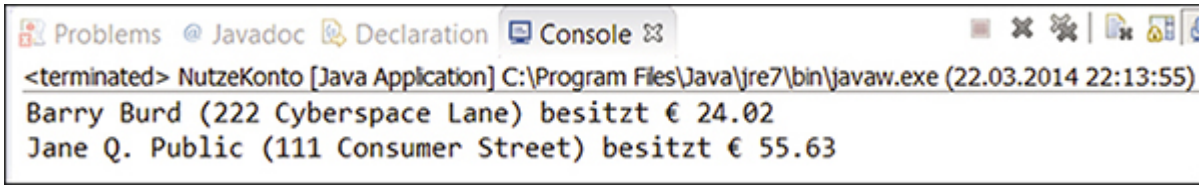
```
Konto meinKonto = new Konto();  
Konto ihrKonto = new Konto();
```

Wenn Sie Zeilen auf diese Weise kombinieren, vermeiden Sie automatisch den Fehler

`Variable might not have been initialized`, den ich im vorherigen Abschnitt beschreibe. Ab und zu stoßen Sie auf eine Situation, in der Sie eine Variable nicht initialisieren können. Sie sollten aber immer daran denken, dass das Initialisieren von Variablen immer ein Plus ist.

Die Felder eines Objekts verwenden

Nachdem Sie die ersten vier Zeilen der Methode `main` analysiert haben, ist der Rest des Codes in [Listing 7.2](#) verständlich und ohne Schnörkel. Es gibt drei Zeilen, die Werte in die Felder des Objekts `meinKonto` schreiben, drei Zeilen, die Werte in die Felder des Objekts `ihrKonto` schreiben, und vier Zeilen, die für die Ausgabe zuständig sind. [Abbildung 7.3](#) zeigt, was das Programm in [Listing 7.2](#) auf dem Bildschirm wiedergibt.



```
<terminated> NutzKonto [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (22.03.2014 22:13:55)
Barry Burd (222 Cyberspace Lane) besitzt € 24.02
Jane Q. Public (111 Consumer Street) besitzt € 55.63
```

Abbildung 7.3: Der Code in [Listing 7.2](#) wird ausgeführt.

Ein Programm; mehrere Klassen

Jedes Programm der [Kapitel 3](#) bis 6 besteht nur aus einer einzigen Klasse. So etwas eignet sich fantastisch für die einleitenden Kapitel eines Buches. Aber im echten Leben besteht ein ganz normales Programm aus Hunderten, wenn nicht sogar aus Tausenden von Klassen. Das Programm, das die Listings 7.1 und 7.2 umfasst, besteht aus zwei Klassen. Natürlich sind zwei Klassen nicht so viel wie Tausende von Klassen, aber es ist ein erster Schritt in diese Richtung.

In der Praxis bringen Programmierer jede Klasse in einer eigenen Datei unter. Wenn Sie ein Programm wie das in den Listings 7.1 und 7.2 erstellen, legen Sie auf der Festplatte Ihres Computers zwei Dateien an. Deshalb erhalten Sie beim Herunterladen der Beispiele dieses Kapitels aus dem Web zwei separate Dateien: `Konto.java` und `NutzKonto.java`.

Öffentliche (»public«) Klassen

Die erste Zeile in [Listing 7.1](#) lautet

```
public class Konto {
```

Dadurch wird die Klasse `Konto` öffentlich (denn das bedeutet `public` auf Deutsch). Eine öffentliche Klasse steht allen anderen Klassen zur Verfügung. Wenn Sie zum Beispiel in einer weit entfernten Ecke des Cyberspace ein Steuerprogramm für Geldautomaten schreiben, kann dieses Programm Code wie `meinKonto.saldo = 24.02` enthalten.

In [Listing 7.2](#) gibt es den Code `meinKonto.saldo = 24.02`. Vielleicht könnten Sie nun glauben, dass die Klasse `Konto` öffentlich sein sollte, weil eine andere Klasse (nämlich die aus

Listing 7.2) die Klasse `Konto` verwendet. Unglücklicherweise ist das alles nicht ganz so einfach, wie es aussieht.

Das schmutzige Geheimnis im Code dieses Kapitels ist, dass ich mich ganz einfach gut fühle, wenn ich Klassen öffentlich deklariere. Ja, auch Programmierer unternehmen bestimmte Dinge nur, um sich gut zu fühlen. In [Listing 7.1](#) hat mein gutes Gefühl seinen Ursprung in der Tatsache, dass eine Klasse `Konto` von vielen Programmierern genutzt werden könnte. Wenn ich eine Klasse deklariere, von der ich glaube, dass sie sinnvoll und nennenswert ist – wie die Klassen `Konto`, `Maschine`, `Kunde`, `GedankenBlitz`, `Kopfschmerzen` oder `KuchenAusSiebenSchichten` –, deklariere ich diese so, dass sie öffentlich (`public`) ist.

Auch die Klasse `NutzKonto` ist in [Listing 7.2](#) öffentlich. Wenn eine Klasse eine `main`-Methode enthält, neigen Java-Programmierer dazu, diese Klasse `public` zu machen und nicht lange darüber nachzudenken, wer sie nutzen könnte. Also deklariere auch ich, ohne zu wissen, ob andere Klassen die Methode `main` verwenden werden, die Klasse `NutzKonto` als `public`. Da die meisten Klassen in diesem Buch eine Methode `main` enthalten, sind sie öffentlich.



Wenn Sie eine Klasse als öffentliche Klasse deklarieren, müssen Sie dies in einer Datei erledigen, deren Name hundertprozentig mit dem Namen der Klasse übereinstimmt (wobei der Dateiname die Erweiterung `.java` erhält). Wenn Sie zum Beispiel den Code `public MeinWichtigerCode` deklarieren, müssen Sie diesen Code in einer Datei mit dem Namen `MeinWichtigerCode.java` ablegen. Dabei müssen Sie unbedingt darauf achten, dass auch die Schreibweise des Namens (Groß-/Kleinschreibung) übereinstimmt. Diese Regel für die Namensgebung hat weitreichende Folgen: Wenn Ihr Code zwei öffentliche Klassen deklariert, muss er aus mindestens zwei `.java`-Dateien bestehen. Oder anders ausgedrückt: Sie können

keine zwei öffentliche Klassen in einer `.java`-Datei deklarieren. ([Kapitel 13](#) enthält weitere Informationen zum Wort `public` und zu anderen in diese Kategorie gehörende Wörter.)

Eine Methode in einer Klasse definieren (ein Konto anzeigen)

Stellen Sie sich als Beispiel eine Tabelle vor, die Informationen über zwei Konten enthält. (Sollten Sie mit einer imaginären Tabelle Schwierigkeiten haben, werfen Sie einen Blick auf [Tabelle 7.1.](#))

Name	Adresse	Saldo
Barry Burd	222 Cyberspace Lane	24.02
Jane Q. Public	111 Consumer Street	55.63

[Tabelle 7.1](#): Ohne objektorientierte Programmierung

In [Tabelle 7.1](#) verfügt jedes Konto über drei Elemente: einen Namen, eine Adresse und einen Saldo. So wurde so etwas gehandhabt, bevor es die objektorientierte Programmierung gab, die dann zu einem großen Umdenken führte. Bei dieser Programmieretechnik kann jedes Konto einen Namen, eine Adresse und einen Saldo haben und angezeigt werden.

Bei der objektorientierten Programmierung verfügt jedes Objekt über eigene »eingebaute« Funktionen. So weiß zum Beispiel ein Konto, wie es sich selbst anzeigen kann. Ein String kann Ihnen sagen, ob es in ihm dieselben Zeichen wie in einem anderen String gibt. Eine `PrintStream`-Instanz wie `System.out` weiß, was es mit `println` auf sich hat. In der objektorientierten Programmierung verfügt jedes Objekt über seine eigenen Methoden. Bei diesen Methoden handelt es sich um kleine Unterprogramme, die Sie aufrufen können, um ein Objekt anzuweisen, bestimmte Aktionen auszuführen.

Und was bringt das? Nun, Sie erreichen, dass Daten die Verantwortung für sich selbst übernehmen. Bei der objektorientierten Programmierung werden alle Funktionen, die zu einem Konto gehören, im Code für die Klasse `Konto` gebündelt. Alles, was Sie über einen String wissen müssen, befindet sich in einer Datei mit dem Namen `String.java`. Alles, was mit Jahreszahlen zu tun hat (ob sie zum Beispiel zwei- oder vierstellig sind), wird in der Klasse `Jahr` behandelt. Wenn nun jemand Probleme mit Ihren Klassen `Konto` oder `Jahr` hat, weiß er, wo er nachsehen muss. Das ist großartig!

Stellen Sie sich nun eine erweiterte Tabelle mit Konteninformationen vor. In dieser neuen Tabelle verfügt jedes Objekt über eingebaute Funktionen. Jedes Konto weiß, wie es sich selbst auf dem Bildschirm anzeigt. Jede Zeile der Tabelle verfügt über eine eigene Version der Methode `display`. Sie müssen nicht viel Vorstellungskraft aufwenden, um diese Tabelle vor Ihren inneren Augen erscheinen zu lassen. Außerdem will es der Zufall, dass hier im Buch eine solche Tabelle abgedruckt worden ist (siehe [Tabelle 7.2](#)).

Name	Adresse	Saldo	Anzeige
Barry Burd	222 Cyberspace Lane	24.02	<code>out.print....</code>
Jane Q. Public	111 Consumer Street	55.63	<code>out.print....</code>

[Tabelle 7.2](#): Die objektorientierte Vorgehensweise

Ein Konto, das sich selbst anzeigt

In [Tabelle 7.2](#) verfügt jedes `Konto`-Objekt über vier Elemente: einen Namen, eine Adresse, einen Saldo und einen Weg, sich selbst auf dem Bildschirm anzuzeigen. Wenn Sie sich einmal für die objektorientierte Programmierung entschieden haben, werden Sie nie wieder auf die alte Weise arbeiten wollen. Die Listings 7.3 und 7.4 stellen Programme dar, die die Ideen aus [Tabelle 7.2](#) umsetzen.

```
import static java.lang.System.out;
public class Konto {
```

```

String name;
String adresse;
double saldo;
public void display() {
    out.print(name);
    out.print(" ");
    out.print(adresse);
    out.print(" besitzt € ");
    out.print(saldo);
}
}

```

[Listing 7.3](#): Ein Konto zeigt sich selbst an.

```

public class NutzKonto {
    public static void main(String args[]) {
        Konto meinKonto = new Konto();
        Konto ihrKonto = new Konto();
        meinKonto.name = "Barry Burd";
        meinKonto.adresse = "222 Cyberspace Lane";
        meinKonto.saldo = 24.02;
        ihrKonto.name = "Jane Q. Public";
        ihrKonto.adresse = "111 Consumer Street";
        ihrKonto.saldo = 55.63;
        meinKonto.display();
        System.out.println();
        ihrKonto.display();
    }
}

```

[Listing 7.4](#): Die verbesserte Klasse *Konto* verwenden

Wenn Sie den Code in den Listings 7.3 und 7.4 ausführen, erhalten Sie das gleiche Ergebnis, das die Listings 7.1 und 7.2 geliefert haben und das [Abbildung 7.3](#) zeigt.

In [Listing 7.3](#) enthält die Klasse `Konto` vier Elemente: die Methoden `name`, `adresse`, `saldo` und `display` (für die Anzeige). Diese vier Elemente entsprechen den vier Spalten in [Tabelle 7.2](#). Damit verfügt jede Instanz der Klasse `Konto` über einen Namen, eine Adresse, einen Saldo und einen Weg, sich selbst anzuzeigen. Und Sie rufen diese Dinge alle einheitlich auf. Um den Namen zu referenzieren, der in `meinKonto` gespeichert ist, schreiben Sie


```
meinKonto.Name
```

Und wenn Sie möchten, dass sich `meinKonto` selbst auf dem Bildschirm wiedergibt, schreiben Sie

```
meinKonto.display()
```

Den einzigen Unterschied bilden die Klammern.



Wenn Sie eine Methode aufrufen, setzen Sie Klammern hinter den Namen der Methode.

Der Kopf der Methode »display«

Schauen Sie sich noch einmal die Listings 7.3 und 7.4 an. In der Methode `main` der Klasse `NutzKonto` gibt es einen Aufruf der Methode `display`. Die Deklaration hat einen Kopf und einen Körper (siehe [Kapitel 3](#)). Der Kopf besteht aus drei Wörtern und einem Klammerpaar:

- ✓ **Das Wort `public` dient ganz grob dem gleichen Zweck wie das Wort `public` in Listing 7.1.** Grob ausgedrückt kann jeder Code den Aufruf einer öffentlichen Methode enthalten. Das geht selbst dann, wenn der aufrufende Code und die öffentliche Methode zu zwei verschiedenen Klassen gehören. Ob das Beispiel in diesem Abschnitt eine öffentliche Methode `display` hat, ist reine Geschmackssache. Wenn ich eine Methode erstelle, die in vielen Anwendungen Verwendung finden kann, deklariere ich diese Methode normalerweise öffentlich (`public`).
- ✓ **Das Wort `void` sagt dem Computer, dass die Methode `display` keinen Wert an den Code zurückgibt, der sie aufruft.** Sie lernen im nächsten Abschnitt eine Methode kennen, die Werte zurückgibt.
- ✓ **Bei dem Wort `display` handelt es sich um den Namen der Methode.** Jede Methode muss einen Namen haben. Anderenfalls gibt es keine Möglichkeit, sie aufzurufen.
- ✓ **Die Klammern enthalten das, was Sie an die Methode übergeben wollen, wenn Sie sie aufrufen.** Wenn Sie eine

Methode aufrufen, können Sie auf die Schnelle Daten an sie übergeben. Die Methode `display` sieht in [Listing 7.3](#) eigenartig aus, weil die Klammern in ihrem Kopf leer sind. Diese Leere weist darauf hin, dass an die Methode `display` nichts übergeben wird, wenn Sie sie aufrufen. Sie lernen im nächsten Abschnitt eine inhaltsschwerere Methode kennen.

[Listing 7.3](#) enthält die Deklaration der Methode `display` und in [Listing 7.4](#) befindet sich der Aufruf dieser Methode. Obwohl beide Listings unterschiedliche Klassen enthalten, wird `public` in [Listing 7.3](#) in beiden Fällen optional verwendet. Wenn Sie erfahren wollen, warum das so ist, sollten Sie sich mit [Kapitel 13](#) beschäftigen.

Werte an Methoden senden und von dort erhalten (Zinsen berechnen)

Stellen Sie sich vor, dass Sie jemanden zum Supermarkt schicken, um Brot zu kaufen. Sie sagen: »Gehe zum Supermarkt und kaufe ein Brot.« Ein anderes Mal schicken Sie die gleiche Person in den Supermarkt, um Bananen zu kaufen. Sie sagen: »Gehe zum Supermarkt und kaufe Bananen.« Und was sagt uns das? Nun, Sie haben eine Methode und Informationen (Daten), die Sie an die Methode übergeben, wenn Sie sie aufrufen. Die Methode heißt *geheZumSupermarktUndKaufe*. Die Daten sind – abhängig von Ihren kulinarischen Bedürfnissen – entweder *Brot* oder *Bananen*. In Java würden die Methodenaufrufe so aussehen:

```
geheZumSupermarktUndKaufe(brot)
geheZumSupermarktUndKaufe(bananen)
```

Das, was in den Klammern steht, wird als *Argumente* oder *Parameterliste* bezeichnet. Wenn Ihre Methoden über Parameter verfügen, sind sie viel vielseitiger einsetzbar. Statt jedes Mal dasselbe zu erhalten, sind Sie nun in der Lage, jemanden zum Supermarkt zu schicken, um Brot, ein anderes Mal Bananen und

dann Vogelfutter zu kaufen. Wenn Sie Ihre Methode `geheZumSupermarktUndKaufe` aufrufen, entscheiden Sie genau dann erst, was gekauft werden soll.

Und was passiert, wenn Ihr Freund vom Supermarkt zurückkehrt? Er sagt dann: »Hier ist das Brot, das ich kaufen sollte.« Indem Ihr Freund Ihrem Wunsch nachgekommen ist, erhalten Sie etwas zurück. Sie haben eine Methode aufgerufen, und die Methode gibt Daten (oder einen frischen Laib Brot) zurück.

Das, was Sie von der Methode zurückerhalten, wird *Rückgabewert* genannt. Der (Daten-)Typ dessen, was zurückgegeben wird, heißt *Rückgabebetyp*. Die Listings 7.5 und 7.6 verdeutlichen diese Konzepte.

```
import static java.lang.System.out;
public class Konto {
    String name;
    String adresse;
    double saldo;
    public void display() {
        out.print(name);
        out.print(" ");
        out.print(adresse);
        out.print(" besitzt € ");
        out.print(saldo);
    }
    public double getZinsen(double zinsSatz) {
        return saldo * zinsSatz / 100.00;
    }
}
```

[Listing 7.5](#): Ein Konto, das selbstständig seine Zinsen berechnet

```
import static java.lang.System.out;
public class NutzKonto {
    public static void main(String args[]) {
        Konto meinKonto = new Konto();
        Konto ihrKonto = new Konto();
        meinKonto.name = "Barry Burd";
        meinKonto.adresse = "222 Cyberspace Lane";
        meinKonto.saldo = 24.02;
        ihrKonto.name = "Jane Q. Public";
        ihrKonto.adresse = "111 Consumer Street";
    }
}
```

```

        ihrKonto.saldo = 55.63;
        meinKonto.display();

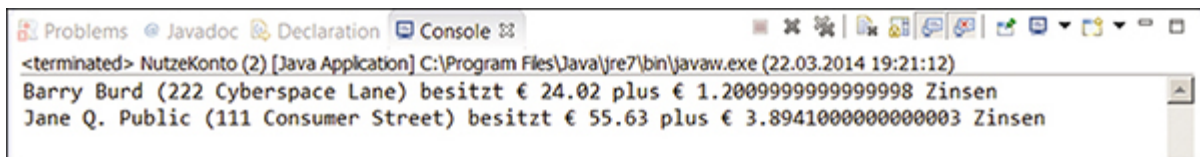
        out.print(" plus € ");
        out.print(meinKonto.getZinsen(5.00));
        out.println(" Zinsen ");
        ihrKonto.display();

        double ihrZinsBetragRate = 7.00;
        out.print(" plus € ");
        double ihrZinsBetrag =
            ihrKonto.getZinsen(ihrZinsSatz);
        out.print(ihrZinsBetrag);
        out.println(" Zinsen ");
    }
}

```

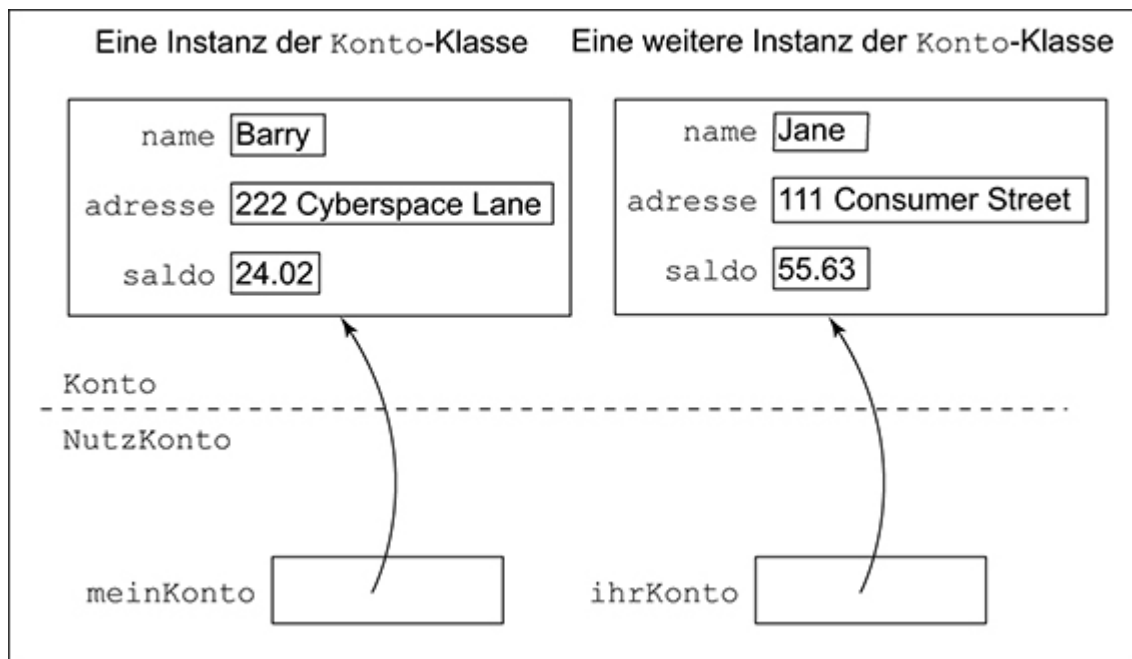
[Listing 7.6](#): Zinsen berechnen

[Abbildung 7.4](#) zeigt, was das Programm in den Listings 7.5 und 7.6 ausgibt, wenn Sie es ausführen. In [Listing 7.5](#) verfügt die Klasse `Konto` über eine Methode mit dem Namen `getZinsen`. Diese Methode wird von der Methode `main` in [Listing 7.6](#) zweimal aufgerufen. Die Kontosalden und die Zinsen sind bei jedem Aufruf anders.



[Abbildung 7.4](#): Der Code in den Listings 7.5 und 7.6 wird ausgeführt.

✓ **Beim ersten Aufruf betragen der Saldo 24.02 und der Zinssatz 5.00.** Der erste Aufruf, `meinKonto.getZinsen(5.00)`, verweist auf das Objekt `meinKonto` und auf die Werte, die in den Feldern dieses Objekts gespeichert sind (siehe [Abbildung 7.5](#)). Wenn dieser Aufruf zum ersten Mal gestartet wird, steht der Ausdruck $\text{saldo} * \text{zinsSatz} / 100.00$ für $24.02 * 5.00 / 100.00$.



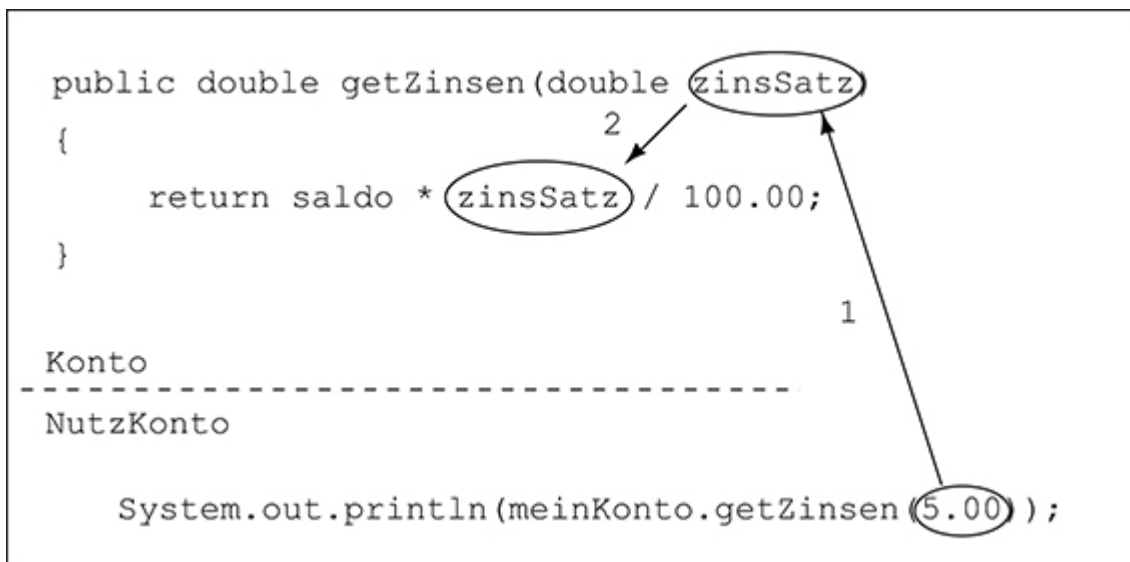
[Abbildung 7.5](#): Mein Konto und Ihr Konto

✓ **Beim zweiten Aufruf betragen der Saldo 55.63 und der Zinssatz 7.00.** In der Methode `main` wird der Variablen `ihrZinsSatz` unmittelbar vor dem zweiten Aufruf der Wert 7.00 zugewiesen. Der Aufruf `ihrKonto.getZinsen(ihrZinsSatz)` verweist auf das Objekt `ihrKonto` und auf die Werte, die in den Feldern dieses Objekts gespeichert sind (siehe noch einmal [Abbildung 7.5](#)). Wenn der Aufruf vorgenommen wird, steht `saldo * zinsSatz / 100.00` für `55.63 * 7.00 / 100.00`.

Nebenbei bemerkt – die Methode `main` in [Listing 7.3](#) enthält zwei Aufrufe von `getZinsen`. Ein Aufruf enthält als Parameter das Literal 5.00; der andere Aufruf enthält als Parameter die Variable `ihrZinsSatz`. Warum verwendet ein Aufruf ein Literal und der andere eine Variable? Ich wollte damit nur zeigen, dass beides möglich ist.

Einen Wert an eine Methode übergeben

Werfen Sie einen Blick auf den Kopf der Methode `getZinsen`. (Während Sie die Erklärungen in der folgenden Liste lesen, können Sie das Konzept, das dahinter steckt, optisch anhand der [Abbildung 7.6](#) verfolgen.)



[Abbildung 7.6](#): Einen Wert an eine Methode übergeben

✓ **Das Wort `double` sagt dem Computer, dass die Methode `getZinsen` bei ihrem Aufruf einen `double`-Wert an den Aufrufer zurückgibt.** Der Befehl im Körper der Methode `getZinsen` bestätigt dies. Er lautet `return saldo * zinsSatz / 100.00`, und auch der Ausdruck `saldo * zinsSatz / 100.00` ist vom Typ `double`. (Der Grund dafür ist, dass alle Elemente des Ausdrucks – `saldo`, `zinsSatz` und `100.00` – vom Typ `double` sind.)

Wenn die Methode `getZinsen` aufgerufen wird, berechnet der Befehl `return saldo * zinsSatz / 100.00` und gibt das Ergebnis an den Code zurück, der die Methode aufgerufen hat.

✓ Bei dem Wort `getZinsen` handelt es sich um den Namen der Methode. Dies ist dann der Name, den Sie verwenden, um die Methode aufzurufen, wenn Sie den Code für die Klasse `NutzKonto` schreiben.

✓ Die Klammern enthalten das, was Sie an die Methode übergeben, wenn Sie sie aufrufen. Wenn Sie eine Methode aufrufen, können Sie auf die Schnelle Daten an sie übergeben. Diese Daten bilden als Argumente die Parameterliste der Methode. Der Kopf der Methode `getZinsen` sagt aus, dass die Methode nur ein Argument haben will, das dann vom Typ `double` sein muss.

```
public double getZinsen(double zinsSatz)
```

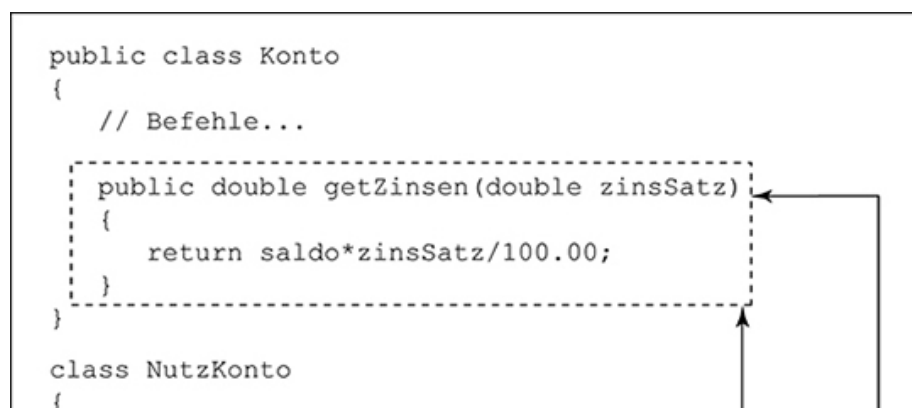
Wenn Sie (in der Methode `main` der Klasse `NutzKonto`) einen Blick auf den ersten Aufruf von `getZinsen` werfen, enthält dieser Aufruf die Zahl `5.00`. Und `5.00` ist ein Literal vom Typ `double`. Wenn ich `getZinsen` aufrufe, gebe ich dieser Variablen einen Wert vom Typ `double` mit. (Falls Sie sich nicht mehr daran erinnern können, was ein Literal ist, siehe [Kapitel 4](#).)

Ähnliches gilt für den zweiten Aufruf von `getZinsen`. Ich rufe gegen Ende von [Listing 7.6](#) `getZinsen` auf und nehme die Variable `ihrZinsSatz` in die Parameterliste der Methode auf. Glücklicherweise war ich so clever, ein paar Zeilen weiter oben `ihrZinsSatz` so zu deklarieren, dass auch dieser Parameter vom Typ `double` ist.

```
public class Konto
{
    // Befehle...

    public double getZinsen(double zinsSatz)
    {
        return saldo*zinsSatz/100.00;
    }
}

class NutzKonto
{
```



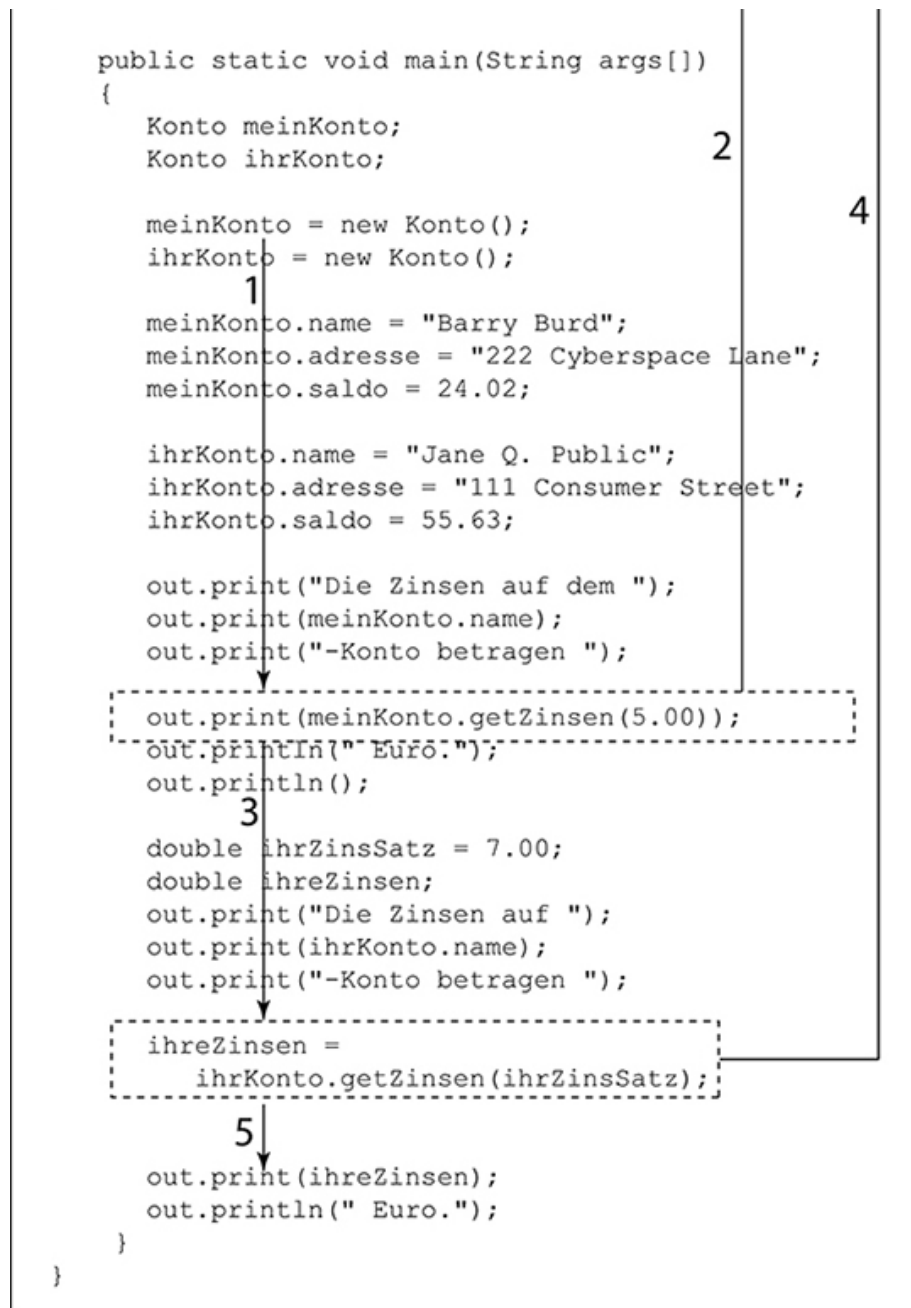


Abbildung 7.7: Der Programmablauf in den [Listings 7.5](#) und [7.6](#)

Wenn Sie den Code in den [Listings 7.5](#) und [7.6](#) ausführen, läuft er nicht geradlinig von oben nach unten ab. Der Programmablauf geht von `main` zu `getZinsen`, dann wieder zurück zu `main` und erneut zu `getZinsen` und zum Schluss wieder zu `main`.

[Abbildung 7.7](#) stellt dies grafisch dar.

Einen Wert von der Methode »getZinsen« zurückgeben

Wenn die Methode `getZinsen` aufgerufen wird, führt sie den `return`-Befehl in ihrem Körper aus. Dieser Befehl berechnet den Wert von `saldo * zinsSatz / 100.00`. Wenn `saldo` 24.02 ist und `zinsSatz` 5.00 beträgt, nimmt der Ausdruck den Wert 1.201 an – was gerundeten 1,20 Euro entspricht. (Da der Computer ausschließlich mit Nullen und Einsen arbeitet, erhält der Computer selbst bei einer so kleinen Zahl ein falsches Ergebnis, nämlich 1.2009999999999998.)

Der Computer führt, nachdem dieser Wert berechnet worden ist, `return` aus, das den Wert an die Stelle von `main` zurücksendet, die `getZinsen` aufgerufen hat. Nun nimmt der gesamte Methodenaufruf – `meinKonto.getZinsen(5.00)` – den Wert 1.2009999999999998 an. Der Aufruf selbst befindet sich innerhalb eines `println`-Befehls:

```
out.println(meinKonto.getZinsen(5.00));
```

Damit nimmt `println` diese Bedeutung an:

```
out.println(1.2009999999999998);
```

[Abbildung 7.8](#) stellt den gesamten Ablauf dar, wie ein Wert an den Methodenaufruf zurückgegeben wird.

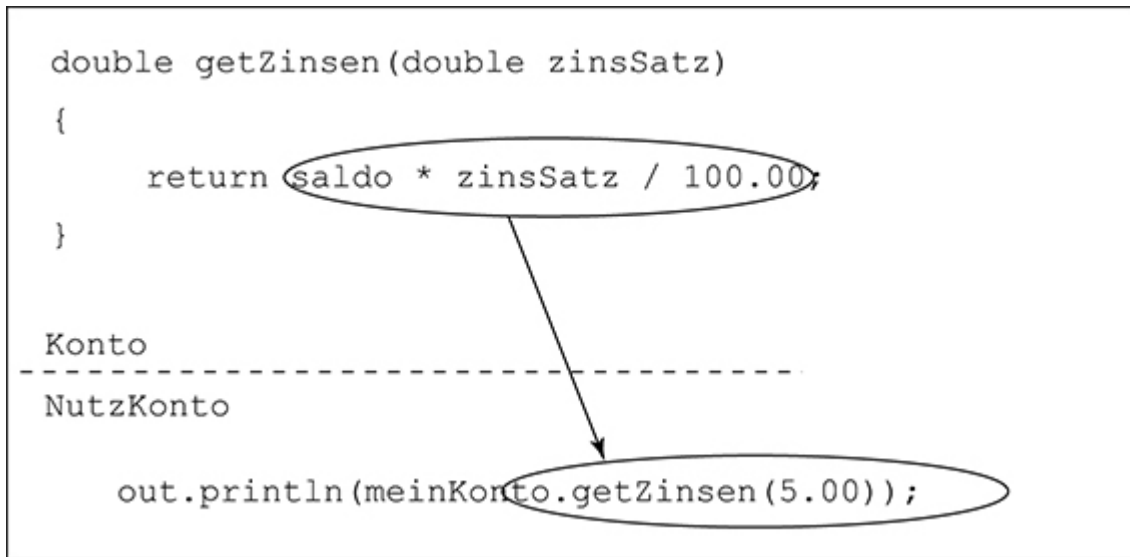


Abbildung 7.8: Ein Methodenaufruf ist ein Ausdruck mit einem Wert.



Wenn eine Methode etwas zurückgibt, handelt es sich bei dem Aufruf der Methode um einen Ausdruck mit einem Wert. Dieser Wert kann ausgegeben, einer Variablen zugewiesen oder zu etwas hinzugefügt werden. Sie können alles, was Sie mit einem Wert machen, auch mit einem Methodenaufruf veranstalten.



Vielleicht verwenden Sie die Klasse `Konto` aus [Listing 7.5](#), um ein echtes Problem zu lösen. Sie könnten die Methoden `display` und `getZinsen` der Klasse `Konto` in einer echten Bankanwendung aufrufen, aber die Klasse `NutzKonto` aus [Listing 7.6](#) sollten Sie für so etwas nicht heranziehen. Der Code dieser Klasse erstellt »getürkte« Kontodaten und ruft dann Methoden der Klasse `Konto` auf, die Ihnen vorgaukeln sollen, dass der Code der Klasse `Konto` auch wirklich funktioniert. (Oder haben Sie ernsthaft daran geglaubt, dass eine Bank Kunden wie *Jane Q. Public* und *Barry Burd* hat?) Die Klasse `NutzKonto` aus [Listing 7.6](#) ist nichts als ein Testfall – eine kurzlebige Klasse, deren einziger Zweck es ist, den Code einer anderen Klasse zu testen. Jeder Testfall in diesem Buch ist wie der Code in [Listing 7.6](#) eine ganz

normale Klasse – eine Freiformklasse mit einer eigenen Methode `main`. Freiformklassen kann man nehmen, aber sie sind eben nicht optimal. Für Java-Programmierer gibt es bessere Möglichkeiten, um Programmtests durchzuführen. Einer der besten davon ist *JUnit*, das unter de.wikipedia.org/wiki/JUnit auf Deutsch und auf der amerikanischen Website des Autors zu diesem Buch unter www.allmycode.com/JavaForDummies passend zum Beispiel der (originalen) Listings beschrieben wird.

Zahlen gut aussehen lassen

Wenn Sie noch einmal einen Blick auf [Abbildung 7.4](#) werfen, irritiert es Sie vielleicht, dass ich nur 1.2009999999999998 Euro Zinsen erhalte. Anscheinend betrügt mich die Bank um zweihundert Trillionstel eines Cents. Eigentlich sollte ich mich unverzüglich beschweren und meinen Zinsanteil einfordern. Vielleicht sollten wir beide uns auf den Weg machen. Wir sollten in der Bank ein wenig Staub aufwirbeln und diesen Betrug öffentlich machen. Wenn meine Vermutung stimmt, kassiert hier jemand richtig ab, indem er von jedem Konto minimale Beträge abzweigt.

Aber halt, warten Sie einen Augenblick! Es gibt keinen Grund, mit mir zur Bank zu gehen. Wenn ich mir noch einmal [Abbildung 7.4](#) ansehe, fällt mir auf, dass Sie viel besser im Spiel sind. Anhand meiner Berechnungen sehe ich, dass das Programm Ihnen dreihundert Trillionstel eines Cents zu viel auszahlt. Wir beide zusammen machen also ein Plus von einhundert Trillionstel eines Cents. Wie das?

Da Computer Nullen und Einsen verwenden und für Berechnungen nicht über unendlich viel Platz verfügen, sind Ungenauigkeiten wie die in [Abbildung 7.4](#) normal. Die schnellste Lösung sieht so aus, dass diese ungenauen Zahlen sinnvoller angezeigt werden. Sie können die Zahlen auf- oder abrunden und nur zwei Nachkommastellen anzeigen lassen. Dabei helfen Ihnen praktische Werkzeuge aus der Java-API (Application

Programming Interface). [Listing 7.7](#) enthält den Code, und [Abbildung 7.8](#) zeigt das erfreuliche Ergebnis.

```
import static java.lang.System.out;
public class NutzKonto {
    public static void main(String args[]) {
        Konto meinKonto = new Konto();
        Konto ihrnKonto = new Konto();
        meinKonto.saldo = 24.02;
        ihrnKonto.saldo = 55.63;
        double meinZinsBetrag = meinKonto.getZinsen(5
.00);
        double ihrZinsBetrag = ihrnKonto.getZinsen(7.
00);

        out.printf("%4.2f EUn", meinZinsBetrag);
        out.printf("%5.2f EUn", meinZinsBetrag);
        out.printf("%.2f EUn", meinZinsBetrag);
        out.printf("%3.2f EUn", meinZinsBetrag);
        out.printf("%.2f EU      %.2f EU",
                    meinZinsBetrag, ihrZinsBetrag);
    }
}
```

[Listing 7.7](#): Zahlen »richtig« aussehen lassen



```
<terminated> NutzKonto (2) [Java A
1,20 EU
 1,20 EU
1,20 EU
1,20 EU
1,20 EU 3,89 EU
```

Abbildung 7.9: Zahlen, die wie Euro-Beträge aussehen



Die ungenauen Zahlen in [Abbildung 7.4](#) rühren daher, wie der Computer Nullen und Einsen verwendet. Ein der Fantasie entsprungener Computer, dessen Schaltkreise so verdrahtet sind, dass er die Ziffern 0, 1, 2, 3, 4, 5, 6, 7, 8 und 9 verwenden kann, würde eine Ungenauigkeit dieser Art nicht

kennen. Um die Ungenauigkeiten von Rechenoperationen eines Computers auszugleichen, verfügt Java über eigene Wege. In Javas API gibt es die Klasse `BigDecimal` – eine Klasse, die die seltsamen Nullen und Einsen des Computers umgeht und für arithmetische Berechnungen ganz normale Dezimalzahlen verwendet. Sie finden eine deutschsprachige Anleitung zur Verwendung von `BigDecimal` unter www.i-coding.de/www/de/java/zahl/bigdecimal-double-float.html, während die (englischsprachige) Java-Dokumentation alle Informationen zu diesem Thema unter docs.oracle.com/javase/1.5.0/docs/api/java/math/BigDecimal.html bereithält.

[Listing 7.7](#) verwendet die praktische Methode `printf`. Wenn Sie `printf` aufrufen, übergeben Sie dem Aufruf in den Klammern immer mindestens zwei Parameter:

✓ **Der erste Parameter ist ein *Formatierungsstring*.**

Der Formatierungsstring verwendet komisch aussehenden Code, um genau zu beschreiben, wie die übrigen Parameter angezeigt werden.

✓ **Bei allen anderen Parametern (die auf den ersten folgen) handelt es sich um die Werte, die ausgegeben werden.**

Schauen Sie sich den letzten `printf`-Aufruf in [Listing 7.7](#) an. Der Formatierungsstring des ersten Parameters weist zwei Platzhalter für Zahlen auf. Der erste Platzhalter (`%.2f`) beschreibt die Anzeige von `meinZinsBetrag`. Der zweite Platzhalter (ein weiteres `%.2f`) beschreibt die Anzeige von `ihrZinsBetrag`. [Abbildung 7.10](#) bis [Abbildung 7.14](#) zeigen an, wie die Formatierungsstrings funktionieren.

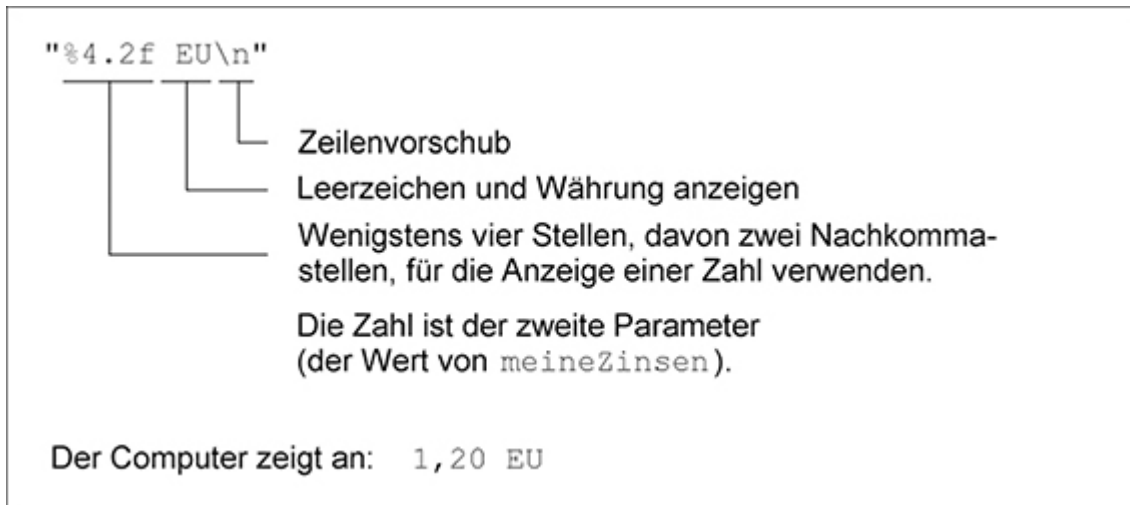


Abbildung 7.10: Einen Formatierungsstring verwenden

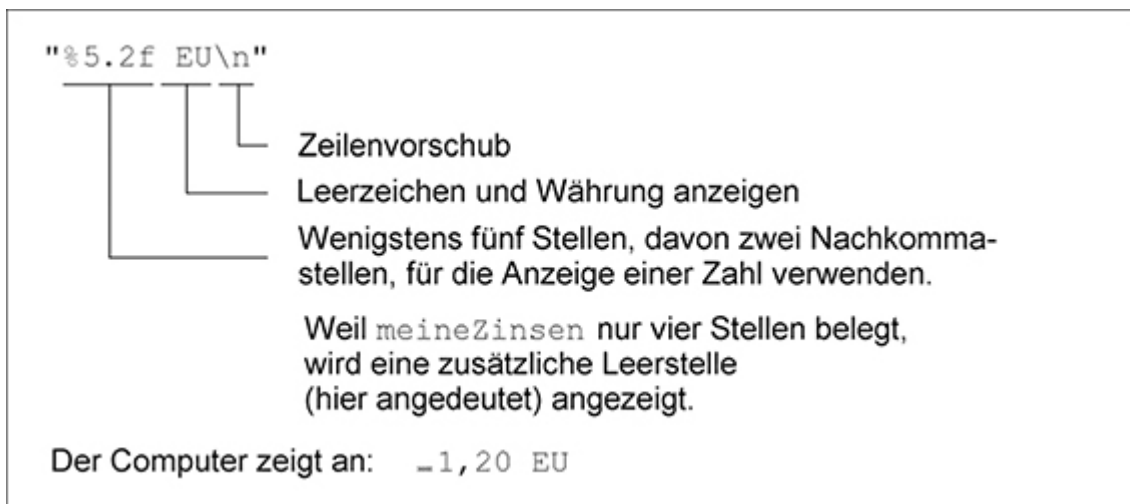


Abbildung 7.11: Leerzeichen hinzufügen, um einen Wert anzuzeigen

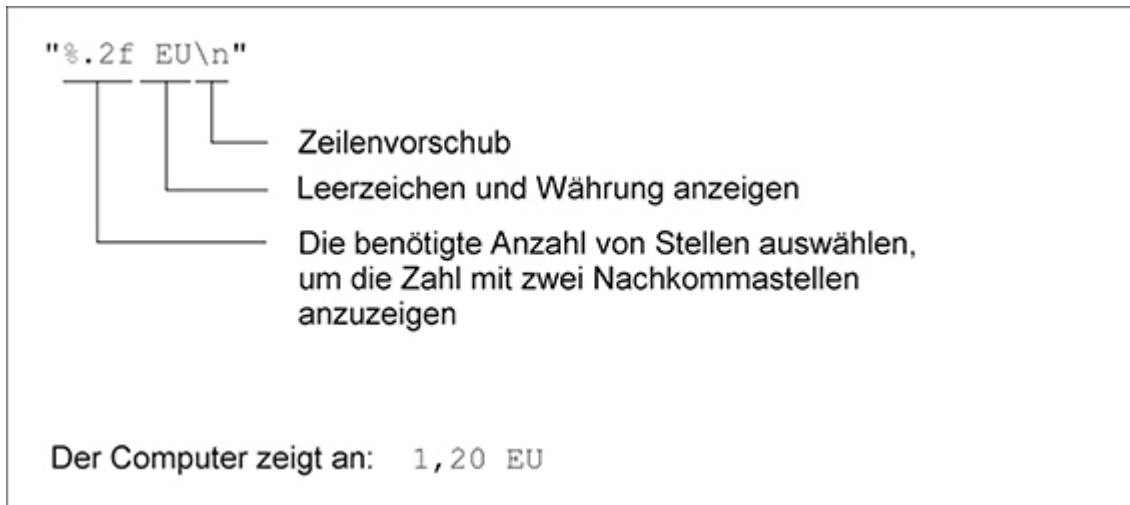


Abbildung 7.12: Einen Wert anzeigen, ohne die genaue Anzahl an Stellen anzugeben

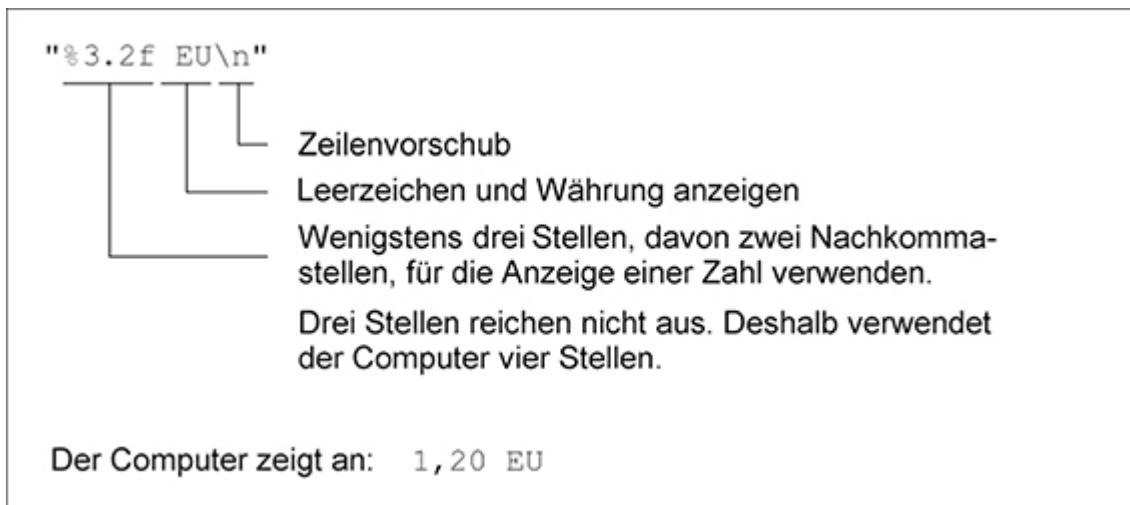
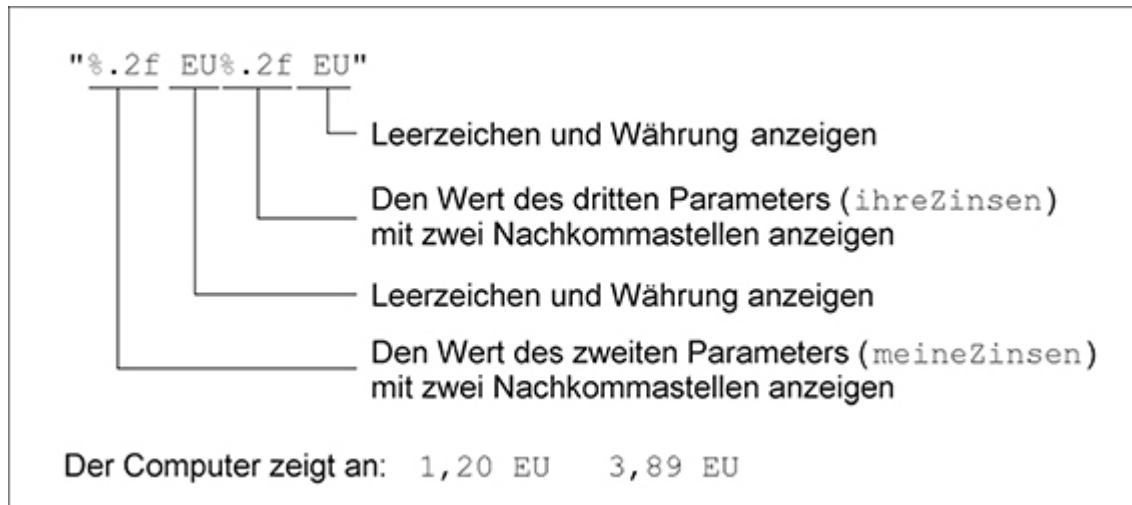


Abbildung 7.13: Zu wenig Stellen für die Anzeige eines Werte festlegen



[Abbildung 7.14](#): Mit einem Formatierungsstring mehr als einen Wert anzeigen

Die [Kapitel 8](#) und 9 enthalten weitere Beispiele für die Verwendung der Methode `printf` und ihrer Formatierungsstrings. Eine Liste aller Möglichkeiten, die die Formatierungsstrings der Methode `printf` bieten, finden Sie auf der Seite `java.util.Formatter` der Dokumentation der Java-API.



Der Formatierungsstring eines `printf`-Aufrufs ändert nichts daran, wie eine Zahl intern für Berechnungen gespeichert wird. Der Formatierungsstring erzeugt nur nett aussehende Ziffern, die auf Ihrem Bildschirm angezeigt werden.

Einzelheiten mit Zugriffsmethoden verbergen

Legen Sie das Buch zur Seite und schnappen Sie sich Ihre Jacke. Sie sind ein so loyaler Leser, dass ich Sie zum Essen einladen möchte.

Ich habe da nur ein Problem. Mir ist mein Bargeld ausgegangen. Macht es Ihnen etwas aus, unterwegs an einem Geldautomaten

anzuhalten und etwas Geld zu ziehen? Wir müssten aber an Ihr Konto gehen, denn auf meinem ist gerade so gut wie nichts drauf.

Glücklicherweise ist es nicht schwer, einen Geldautomaten zu bedienen. Gehen Sie einfach hin, stecken Sie Ihre Bankkarte in die Maschine und geben Sie Ihre PIN ein. Danach fragt Sie der Automat, unter welchem Variablennamen Sie Ihren aktuellen Saldo speichern wollen. Sie können unter `saldo`, `saldo324`, `meinSal`, `aktuellerSaldo`, `s€`, `SALDO`, `asj999` und `konStanTinopel` wählen. Nachdem Sie einen Variablennamen gewählt haben, sollten Sie einen Speicherplatz für den Wert der Variablen aussuchen. Sie können jede Zahl zwischen `022FFF` und `0555AA` nehmen. (Diese Zahlen liegen im hexadezimalen Format vor.) Nachdem Sie die Software des Bankautomaten konfiguriert haben, sollte es kein Problem mehr sein, an Bargeld zu kommen.

Gute Programmierung

Ein guter Programmierstil lässt sich in einem Wort zusammenfassen: *Einfachheit*. Wenn Sie ein kompliziertes Programm schreiben, haben Sie keine Lust, sich auch noch mit falsch benannten Variablen, geschraubten Problemlösungen oder cleveren Programmiertricks herumschlagen zu müssen, die anderen Programmierern auf den letzten Drücker eingefallen sind. Sie wollen mit einer sauberen Schnittstelle arbeiten, mit der Sie Ihre und nicht die Probleme anderer Leute lösen können.

In dem Szenario mit dem Geldautomaten, das ich oben beschreibe, besteht das Problem darin, dass der Apparat Sie zwingt, sich mit anderer Leute Dinge zu beschäftigen. Statt schnell an Ihr Geld für das Mittagessen zu kommen, müssen Sie über Variablen und Speicherorte nachdenken. Es ist klar, dass sich jemand um die technischen Probleme des Geldautomaten kümmern muss – aber das ist nicht Ihre Aufgabe als Kunde der Bank.



In diesem Abschnitt geht es um Bedienungsfreundlichkeit, nicht um Sicherheit. Bedienungsfreundlicher Code verhindert, dass Sie aus Versehen Programmierfehler machen. Sicherer Code (bei dem es sich um eine ganz andere Baustelle handelt) hält bössartige Hacker davon ab, Schäden anzurichten.

Muss also alles, was einen Bezug zu einem Computerprogramm hat, einfach gehalten werden? Nein. Manchmal müssen Sie viel an Vorbereitungszeit in eine Sache stecken, damit sie im Endeffekt einfach zu bedienen ist. Die Menschen, die den Geldautomaten gebaut haben, haben hart gearbeitet, damit der Apparat benutzerfreundlich wird. Die Schnittstelle der Maschine mit ihren Nachrichten auf dem Bildschirm und den Schaltflächen machen aus dem Automaten ein kompliziertes, aber sorgfältig entworfenes Gerät.

Die Aufgabe, etwas einfach aussehen zu lassen, erfordert sorgfältige Planung. Im Fall der objektorientierten Programmierung besteht einer der Wege, Dinge einfach aussehen zu lassen, darin, Code, der außerhalb einer Klasse definiert wird, daran zu hindern, direkt auf Felder zuzugreifen, die in der Klasse angelegt werden. Werfen Sie noch einmal einen Blick auf den Code in [Listing 7.1](#). Sie arbeiten in einem Unternehmen, das gerade zehn Millionen für den Code in der Klasse `Konto` bezahlt hat. (Das sind mehr als eineinhalb Millionen Euro pro Zeile.) Jetzt haben Sie die Aufgabe, die Klasse `NutzKonto` zu schreiben. Sie möchten

```
meinKonto.name = "Barry Burd";
```

schreiben, aber damit würden Sie zu tief in die Klasse `Konto` eingreifen. Schließlich dürfen Personen, die einen Geldautomaten nutzen, nicht die Variablen des Programms programmieren. Sie dürfen die Tastatur des Automaten nicht benutzen, um den Befehl

```
saldoDesKontos29872865457 =  
    saldoDesKontos29872865457 + 1000000.00;
```

einzugeben. Stattdessen drücken sie Tasten und Schaltflächen, die diese Aufgabe in einer vorgegebenen Reihenfolge erledigen.

Und genau auf diese Weise erreichen Programmierer
Bedienerfreundlichkeit und Einfachheit.

Damit alles pflegeleicht und ordentlich abläuft, müssen Sie die
Klasse `Konto` aus [Listing 7.1](#) ändern, indem Sie die Zeilen

```
meinKonto.name = "Barry Burd";  
und
```

```
out.print(ihrKonto.saldo);
```

entfernen. Dies sorgt wiederum für ein Problem. Sie sind
derjenige, der den Code für die Klasse `NutzKonto` schreibt.
Wenn Sie `meinKonto.name` oder `ihrKonto.saldo` nicht
verwenden dürfen, wie sollen Sie dann Ihr Ziel erreichen? Die
Antwort besteht aus etwas, das *Zugriffsmethoden* heißt. Die
Listings 7.8 und 7.9 führen den Umgang mit diesen Methoden vor.

```
public class Konto {  
    private String name;  
    private String adresse;  
    private double saldo;  
    public void setName(String n) {  
        name = n;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setAdresse(String a) {  
        adresse = a;  
    }  
    public String getAdresse() {  
        return adresse;  
    }  
    public void setSaldo(double s) {  
        saldo = s;  
    }  
    public double getSaldo() {  
        return saldo;  
    }  
}
```

[Listing 7.8](#): Die Felder verbergen

```
import static java.lang.System.out;  
public class NutzKonto {
```

```

public static void main(String args[]) {
    Konto meinKonto = new Konto();
    Konto ihrKonto = new Konto();
    meinKonto.setName("Barry Burd");
    meinKonto.setAdresse("222 Cyberspace Lane");
    meinKonto.setSaldo(24.02);
    ihrKonto.setName("Jane Q. Public");
    ihrKonto.setAdresse("111 Consumer Street");
    ihrKonto.setSaldo(55.63);
    out.print(meinKonto.getName());
    out.print(" ");
    out.print(meinKonto.getAdresse());
    out.print(" besitzt € ");
    out.print(meinKonto.getSaldo());
    out.println();
    out.print(ihrKonto.getName());
    out.print(" ");
    out.print(ihrKonto.getAdresse());
    out.print(" besitzt € ");
    out.print(ihrKonto.getSaldo());
}
}

```

[Listing 7.9](#): Zugriffsmethoden aufrufen

Wenn Sie das Programm in den [Listing 7.8](#) und [7.9](#) ausführen, unterscheidet sich das Ergebnis nicht von dem, was Sie von den [Listings 7.1](#) und [7.2](#) her kennen. [Abbildung 7.3](#) zeigt, was beide Programme ausgeben. Der große Unterschied liegt darin, dass die Klasse `Konto` in [Listing 7.8](#) die Verwendung ihrer Felder `name`, `adresse` und `saldo` sorgfältig kontrolliert.

Öffentliches Leben und private Träume: ein Feld sperren

Die Deklarationen der Felder der Klasse `Konto` sind um das Wort `private` erweitert worden. `private` ist ein Java-Schlüsselwort. Wenn ein Feld zu einem privaten Feld gemacht wird, kann kein Code das Feld von außerhalb der Klasse direkt referenzieren. Wenn Sie dann in [Listing 7.9](#) in der Klasse `NutzKonto` festlegen, dass `meinKonto.name = "Barry Burd"` ist, erhalten Sie die

Fehlermeldung `name has private access in Konto`, die aussagt, dass das Feld privat deklariert wurde.

Statt nun auf `meinKonto.name` zu verweisen, muss der Programmierer von `NutzKonto` die Methode `meinKonto.setName` oder die Methode `meinKonto.getName` aufrufen. Diese Methoden werden als *Zugriffsmethoden* bezeichnet, weil sie dafür sorgen, dass auf das Feld `name` der Klasse `Konto` zugegriffen werden kann. (Dabei sollten Sie aber wissen, dass der Begriff *Zugriffsmethode* kein Bestandteil der offiziellen Java-Terminologie ist. Er wird einfach nur von Programmierern für diese Art von Methoden verwendet.) Noch spezieller wird `setName` als *Setter-* (Setzer-) oder *Set-Methode* und `getName` als *Getter-* (Holer-) oder *Get-Methode* bezeichnet. (Ich wette, dass Sie diese Begriffe nicht mehr vergessen!)



Bei vielen IDEs müssen Sie keine eigenen Zugriffsmethoden schreiben. Geben Sie als Erstes die Deklaration des Feldes ein (zum Beispiel `private Stringname`). Wählen Sie dann in der Menüleiste Ihrer IDE `SOURCE|GENERATE GETTER AND SETTER` oder `CODE|INSERT CODE|SETTER` oder einen ähnlichen Befehl. Nachdem Sie Ihre Wahl getroffen haben, erstellt die IDE Zugriffsmethoden und fügt sie Ihrem Code hinzu.

Beachten Sie, dass die Setter- und Getter-Methoden in [Listing 7.8](#) als öffentliche (`public`) Methoden deklariert werden. Damit erreichen Sie, dass jeder diese beiden Methoden von überall her aufrufen kann. Dadurch wird es zwar unmöglich, die Felder von außerhalb des `Konto`-Codes zu ändern, aber die Setter- und Getter-Methoden können problemlos darauf zugreifen, um die entsprechenden Felder zu nutzen. (Sie erfahren in [Kapitel 13](#) mehr über die Schlüsselwörter `public` und `private`.)

Denken Sie noch einmal an den Bankautomaten. Niemand, der den Automaten benutzt, ist in der Lage, einen Befehl einzugeben, der direkt den Saldo seines Kontos ändert, aber es ist leicht, einen Scheck über eine Million Euro einzureichen. Die Leute, die

den Bankautomaten entwickelt haben, wissen, dass es viele Probleme gegeben hätte, wenn dieser Vorgang kompliziert gewesen wäre. Damit lässt sich zusammenfassend sagen, dass Sie alles verhindern müssen, was Benutzer nicht dürfen, und dass im Gegenzug alle Aufgaben, die getan werden müssen, auch leicht zu erledigen sind.



Setter- und Getter-Methoden sind nichts Weltbewegendes. Sie müssen nur die Setter- und Getter-Methoden schreiben, die Sie benötigen. So kann ich zum Beispiel in [Listing 7.8](#) auf die Deklaration der Methode `getAdresse` verzichten, ohne dass dies den Programmablauf beeinflussen würde. Es könnte dann nur zu dem Problem kommen, dass jemand, der meine `Konto`-Klasse verwenden und die Adresse eines Kontos erhalten möchte, völlig aufgeschmissen ist.



Wenn Sie eine Methode erstellen, die den Wert eines `saldo`-Feldes setzt, müssen Sie diese Methode nicht `setSaldo` nennen. Sie können ihr einen beliebigen Namen wie zum Beispiel `thunFisch` geben. Aber mittlerweile hat sich weltweit eine stilistische Konvention mit `setFeldname` durchgesetzt (bei der `set` mit einem kleinen Anfangsbuchstaben und `Feldname` mit einem Großbuchstaben geschrieben werden). Eine individuelle Namensgebung oder Schreibweise würde nur zur Verwirrung anderer Java-Programmierer führen. Wenn Ihre Entwicklungsumgebung (IDE) über die Möglichkeit verfügt, eine GUI mit der Maus zu entwerfen, kann Ihre individuelle Namensgebung dazu führen, dass Ihre IDE diese Fähigkeit zeitweilig verliert. (Java-Entwicklungsumgebungen und *Drag-and-Drop* werden in den [Kapiteln 2](#) und [14](#) behandelt.)



Wenn Sie eine Setter-Methode aufrufen, übergeben Sie ihr einen Wert, der dem definierten Typ entspricht. Deshalb rufen Sie in [Listing 7.9](#) `ihrKonto.setSaldo (55.63)` auch mit

einem Parameter vom Typ `double` auf. Im Gegensatz dazu rufen Sie eine Getter-Methode normalerweise ohne Werte auf. Deshalb enthält zum Beispiel in [Listing 7.9](#) `ihrKonto.getSaldo()` eine leere Parameterliste. Gelegentlich kommt es vor, dass Sie einen Wert mit einem einzelnen Befehl holen oder setzen wollen. Um dem Saldo Ihres Kontos einen Euro hinzuzufügen, schreiben Sie `ihrKonto.setSaldo(ihrKonto.getSaldo() + 1.00)`.

Regeln mit Zugriffsmethoden erzwingen

Beschäftigen wir uns noch einmal mit [Listing 7.8](#). Werfen Sie dort einen Blick auf die Methode `setName`. Stellen Sie sich vor, dass Sie die Zuweisungsanweisung der Methode in eine `if`-Anweisung packen.

```
public void setName(String n) {  
    if (!n.equals("")) {  
        name = n;  
    }  
}
```

Wenn nun der Programmierer, der mit der Klasse `NutzKonto` beschäftigt ist, `meinKonto.setName("")` schreibt, zeigt der Aufruf von `setName` keine Wirkung mehr. Da das Feld `name` privat ist, wird der folgende Aufruf in der Klasse `NutzKonto` unzulässig:

```
meinKonto.name = "";
```

Natürlich funktioniert ein Aufruf wie

`meinKonto.setName("Willi Schmitz")` auch weiterhin, weil `"Willi Schmitz"` nicht gleich dem leeren String `""` ist.

Das ist stark. Sie können mit einem privaten Feld und einer Zugriffsmethode verhindern, dass jemand dem Feld `name` eines Kontos einen leeren String zuweist. Sie sehen, dass Sie mit sauber ausgearbeiteten `if`-Befehlen so gut wie jede Art von Regel erzwingen können.

8

Zeit und Geld sparen: Code wiederverwenden

In diesem Kapitel

- ▶ Alten Code neu beleben
 - ▶ Code optimieren
 - ▶ Änderungen vornehmen, ohne ein Vermögen auszugeben
-

Vor langer Zeit lebte in einem fernen Land eine wunderschöne Prinzessin. Als die Prinzessin 25 Jahre alt wurde (das optimale Alter für Macht, gutes Aussehen und hohe moralische Ansprüche), erhielt sie von ihrem gütigen Vater ein Geschenk in einer goldenen Schachtel. Neugierig, wie sie war, riss die Prinzessin die goldene Verpackung auf.

Die Prinzessin war ganz aufgeregt. Die Überraschung war ihrem Vater gelungen, denn er hatte ihr das geschenkt, was sie sich schon immer gewünscht hatte – ein fehlerfrei laufendes Computerprogramm. Das Programm machte alles, was die Prinzessin von ihm verlangte, und das so, wie sie es erwartete. Die Prinzessin war glücklich, und das war auch ihr gütiger, alter Vater.

Über viele Jahre hinweg lief das Programm fehlerfrei. Aber dann änderten sich die Bedürfnisse der Prinzessin. Sie erwartete mehr vom Leben, stellte höhere Ansprüche, kümmerte sich mehr um ihre Karriere, strebte nach immer größerer Erfüllung, jonglierte mit den Wünschen ihres Ehemannes und ihrer Kinder, nahm es mit ihrem Budget nicht so genau und suchte Frieden für ihre Seele. Und immer war das Programm ihr treuer Begleiter.

Die Prinzessin und das Programm wurden zusammen alt. Eines Abends, als sie am Kamin saß, stellte sie dem Programm eine beängstigende Frage: »Wie machst du das? Wie schaffst du es, Jahr für Jahr die richtigen Antworten zu geben?«

»Gesundes Leben«, antwortete das Programm. »Ich schwimme jeden Tag 20 Apps, und ich nehme C++ gegen Viren. Ich vermeide hogarithmische Algorithmen. Ich linke Java in Maßen. Ich sage GNU zu Bugs. Ich rauche keine Backups. Und ich byte nie mehr ab, als ich vertrage.«

Die Prinzessin war sprachlos.

Eine Klasse definieren (was es bedeutet, ein Mitarbeiter zu sein)

Wäre es nicht ganz nett, wenn jedes Stückchen Software das täte, was Sie von ihm erwarten? Sie können in einer idealen Welt einfach ein Programm kaufen, es betriebsbereit machen, es nahtlos an neue Situationen anpassen und es problemlos aktualisieren, wenn das notwendig wird. Unglücklicherweise gibt es keine Software dieser Art. (*Nichts* dieser Art existiert.) Die Wahrheit ist, dass Sie Software finden können, die einiges von dem kann, was Ihnen so vorschwebt, aber eben nicht alles.

Dies ist einer der Gründe dafür, dass objektorientierte Programmierung so erfolgreich geworden ist. Seit Jahren erwerben Unternehmen vorgefertigten Code, um dann festzustellen, dass dieser Code nicht das macht, was er ihrer Meinung nach tun sollte. Die Unternehmen begannen also, am Code herumzubasteln. Ihre Programmierer tauchten immer tiefer in die Programmdateien ein, änderten Namen von Variablen, umgaben alles mit Unterprogrammen, bauten Formulare neu auf und schafften es im Allgemeinen nur, den Code zu verschlimmbessern. Die Wirklichkeit sieht eben so aus, dass Sie

ein Programm, das nicht das macht, was Sie wollen, nicht dadurch verbessern, dass Sie an seinem Code herumbasteln. Das gelingt Ihnen selbst dann nicht, wenn das Programm schon fast alles kann, von dem Sie träumen. Die beste Alternative sieht dann so aus, das Programm (unabhängig davon, wie teuer es war) wegzuworfen und ganz neu von vorn anzufangen. Was für ein schlechtes Geschäft!

Objektorientierte Programmierung hat dies gewaltig geändert. Ein objektorientiertes Programm ist in seinem Kern so entworfen worden, dass es geändert werden kann. Mit einer sauber geschriebener Software sind Sie in der Lage, Nutzen aus Funktionen zu ziehen, die bereits vorhanden sind, neue Funktionen selbstständig hinzuzufügen und Funktionen zu überschreiben, die Ihren Bedürfnissen nicht entsprechen. Das Beste an dieser Situation ist, dass die Änderungen, die Sie vornehmen, sauber sind – kein Kratzen und Graben am beziehungsweise im empfindlichen Programmcode anderer Entwickler. Stattdessen fügen Sie sorgfältig und ordentlich Erweiterungen und Änderungen hinzu, ohne die interne Logik des vorhandenen Codes anzurühren.

Das letzte Wort an Mitarbeiter

Wenn Sie ein objektorientiertes Programm schreiben, beginnen Sie damit, dass Sie sich Gedanken über die zu verarbeitenden Daten machen. Sie schreiben über Konten. Also, was ist ein Konto? Sie schreiben Code, der auf das Anklicken einer Schaltfläche reagiert. Also, was ist eine Schaltfläche? Sie schreiben ein Programm, um Gehaltsschecks an Mitarbeiter zu versenden. Also, was ist ein Mitarbeiter?

Im ersten Beispiel dieses Kapitels ist ein Mitarbeiter eine Person mit einem Namen und einer Funktion im Unternehmen. Natürlich haben Mitarbeiter auch noch andere Merkmale, aber im Moment wollen wir uns auf Grundlegendes konzentrieren. Der Code in [Listing 8.1](#) beschreibt, was es heißt, ein Mitarbeiter zu sein.

```
import static java.lang.System.out;  
public class Mitarbeiter {
```

```

private String name;
private String funktion;
public void setName(String nameIn) {
    name = nameIn;
}
public String getName() {
    return name;
}
public void setFunktion(String funktionIn) {
    funktion = funktionIn;
}
public String getFunktion() {
    return funktion;
}
public void zahleGehalt(double betrag) {
    out.printf("Zahlen Sie an %s ", name);
    out.printf("(%s) ***", funktion);
    out.printf("%,.2f Euron", betrag);
}
}

```

[Listing 8.1](#): Was ist ein Mitarbeiter?

Anhand von [Listing 8.1](#) können Sie sehen, dass jeder Mitarbeiter über sieben Merkmale definiert wird. Zwei dieser Merkmale sind ziemlich einfach: Jeder Mitarbeiter hat einen Namen und eine Funktion im Unternehmen. (Die Klasse `Mitarbeiter` verfügt in [Listing 8.1](#) über ein Feld `name` und ein Feld `funktion`.)

Und was hat ein Mitarbeiter sonst noch zu bieten? Zum Beispiel vier Methoden, um mit den Werten für den Namen und die Funktion des Mitarbeiters umgehen zu können. Bei diesen Methoden handelt es sich um `setName`, `getName`, `setFunktion` und `getFunktion`. Ich erkläre diese Art von Methoden (*Zugriffsmethoden*) in [Kapitel 7](#).

Und dann enthält die Klasse noch die Methode `zahleGehalt`. Ich habe mir dabei überlegt, dass die Methode, die für die Auszahlung des Gehalts zuständig ist, zu einer der beiden Klassen gehören muss. Da der größte Teil der Daten, die zu einer Gehaltszahlung gehören, für die einzelnen Mitarbeiter aufbereitet

sind, befindet sich die Methode `zahleGehalt` in der Klasse `Mitarbeiter`.

Einzelheiten zum Aufruf von `printf` in der Methode `zahleGehalt` finden Sie weiter hinten in diesem Kapitel im Abschnitt *Gehalt bezahlen*.

Die Klasse gut verwenden

Die Klasse `Mitarbeiter` verfügt in [Listing 8.1](#) nicht über eine Methode `main`. Damit gibt es bei der Ausführung des Codes keinen Punkt, von dem aus das Programm starten kann. Um diese Schwachstelle zu beheben, schreiben Sie ein separates Programm, das eine Methode `main` enthält. Verwenden Sie dann dieses Programm, um Instanzen von `Mitarbeiter` zu erstellen. [Listing 8.2](#) enthält eine Klasse mit einer Methode `main`.

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
public class Gehaltsabrechnung {
    public static void main(String args[])
                                throws IOException
    {
        Scanner diskScanner =
            new Scanner(new File("EmployeeInfo.txt"))
;
        for (int empNum = 1; empNum <= 3; empNum++) {
            bezahleMitarbeiter(diskScanner);
        }
        diskScanner.close();
    }
    static void bezahleMitarbeiter(Scanner aScanner)
    {
        Mitarbeiter einMa = new Mitarbeiter();
        einMa.setName(aScanner.nextLine());
        einMa.setFunktion(aScanner.nextLine());
        einMa.zahleGehalt(aScanner.nextDouble());
        aScanner.nextLine();
    }
}
```

[Listing 8.2](#): Eine Gehaltsanweisung fertigmachen



Um den Code in [Listing 8.2](#) auszuführen, muss es auf Ihrer Festplatte eine Datei mit dem Namen `EmployeeInfo.txt` geben. Für die Eclipse-IDE existiert das Beispielprojekt 08-01. Dieses Projekt befindet sich auf meiner Festplatte im Ordner `c:Benutzermein-Benutzernameworkspace08-01`. Und in diesem Ordner gibt es die Datei `EmployeeInfo.txt`. Wenn Sie mehr über Dateien auf Ihrer Festplatte nachlesen wollen, schauen Sie sich weiter hinten in diesem Kapitel den Abschnitt *Mit Dateien auf der Festplatte arbeiten (ein kleiner Umweg)* an.

Die Klasse `Gehaltsabrechnung` verfügt in [Listing 8.2](#) über zwei Methoden. Eine davon, die Methode `main`, ruft die Methode `bezahleMitarbeiter` dreimal auf. Dabei erhält diese Methode jedes Mal Daten von der Datei `EmployeeInfo.txt` und füttert damit die Methoden der Klasse `Mitarbeiter`.

Und so wird die Variable `bezahleMitarbeiter` wiederverwendet:

- ✓ Wenn `bezahleMitarbeiter` zum ersten Mal aufgerufen wird, sorgt sie dafür, dass im Befehl `Mitarbeiter einMa = new Mitarbeiter() einMa` auf ein neues Objekt verweist.
- ✓ Beim zweiten Aufruf von `bezahleMitarbeiter` führt der Computer denselben Befehl erneut aus, wodurch es zu einer neuen Verkörperung der Variablen `einMa` kommt, die ein brandneues Objekt erstellt.
- ✓ Beim dritten Durchlauf geschieht dies alles noch einmal. Es kommt zu einer neuen Variablen `einMa`, die auf ein drittes Objekt verweist.

[Abbildung 8.1](#) stellt diesen Ablauf grafisch dar.

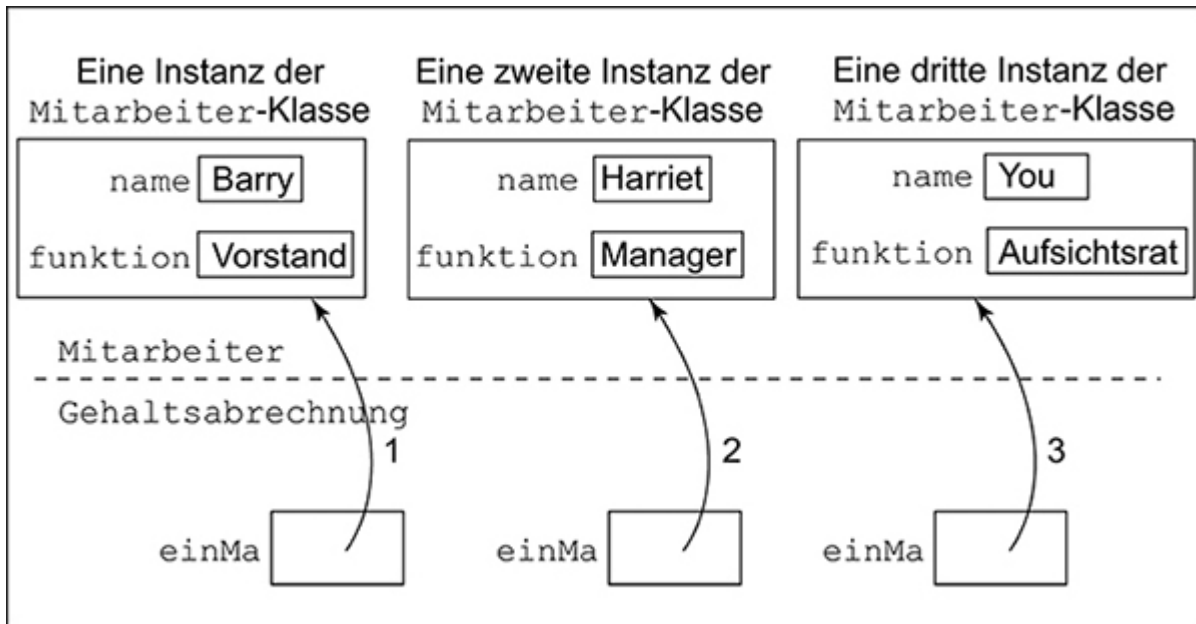
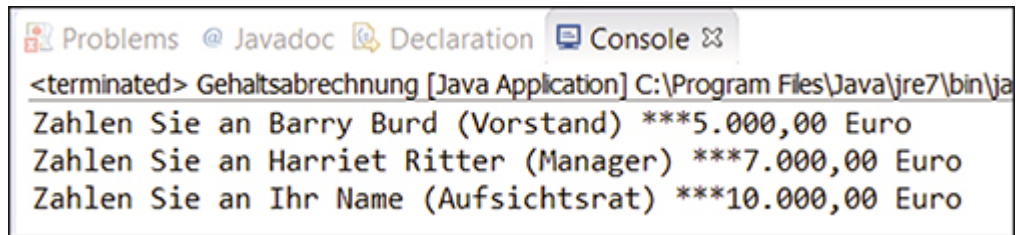


Abbildung 8.1: Drei Aufrufe der Methode *bezahleMitarbeiter*

Gehalt bezahlen

[Listing 8.1](#) enthält drei `printf`-Aufrufe. Jeder dieser Aufrufe verfügt über einen Formatierungsstring (wie "`(%s) ****`") und eine Variable (wie `funktion`). Jeder Formatierungsstring enthält einen Platzhalter (wie `%s`), der festlegt, wo und wie der Wert der Variablen angezeigt wird.

So enthält zum Beispiel der Formatierungsstring im zweiten Aufruf von `printf` einen Platzhalter `%s`, der einen Platz für den Wert der Variablen `funktion` frei hält. Die Schreibweise `%s` dient entsprechend den Java-Regeln als Platzhalter für einen String, und `funktion` ist in [Listing 8.2](#) als `String` definiert worden. Der Platzhalter `%s` wird von Klammern und ein paar anderen Zeichen umgeben, was letztendlich dazu führt, dass auch die berufliche Position des Mitarbeiters (in den Beispielen dieses Kapitels *Funktion* genannt) bei der Ausgabe in Klammern steht (siehe [Abbildung 8.2](#)).



```
<terminated> Gehaltsabrechnung [Java Application] C:\Program Files\Java\jre7\bin\ja
Zahlen Sie an Barry Burd (Vorstand) ***5.000,00 Euro
Zahlen Sie an Harriet Ritter (Manager) ***7.000,00 Euro
Zahlen Sie an Ihr Name (Aufsichtsrat) ***10.000,00 Euro
```

[Abbildung 8.2](#): Jeder wird bezahlt.

Beachten Sie in [Listing 8.1](#) das Komma beim Platzhalter `%`, `.2`. Das Komma weist das Programm an, Zahlen zu gruppieren, weshalb so etwas auch *gruppierender Separator* genannt wird. Aus diesem Grund sehen Sie in [Abbildung 8.2](#) nicht `5000.00` Euro, `7000.00` Euro und `10000.00` Euro, sondern `5.000,00` Euro, `7.000,00` Euro und `10.000,00` Euro.



Gruppierende Separatoren unterscheiden sich von Land zu Land. So schreiben Sie zum Beispiel in Deutschland eintausend `1.000,00`. Java kann Ihre Zahlen automatisch über einen Befehl wie `out.print(new java.util.Formatter().format(java.util.Locale.GERMANY, "%,.2f", 1000.00))` deutsch darstellen. Einzelheiten hierzu finden Sie in der Dokumentation der Java-API zu den Klassen `Formatter` und `Locale`.

Mit Dateien auf der Festplatte arbeiten (ein kleiner Umweg)

In [Kapitel 7](#) lesen Programme Zeichen, die über die Tastatur des Computers eingegeben werden. Im Gegensatz dazu liest der Code in [Listing 8.2](#) Zeichen, die aus einer Datei stammen. Diese Datei, `MitarbeiterInfo.txt`, liegt auf der Festplatte Ihres Computers.

Die Datei entspricht dem Dokument einer Textverarbeitung, wobei sie aber keine Formatierungen enthält – nichts Kursives, keine Fettschrift, keine Schriftgröße, nichts dieser Art.

Die Datei `MitarbeiterInfo.txt` enthält nur Standardzeichen – die Art von Tastatureingaben, zu denen es kommt, wenn Sie zum Beispiel das Ratespiel aus [Kapitel 5](#) oder 6 spielen. Natürlich sind das Auslesen von Tastatureingaben während eines Ratespiels und das Auslesen einer Datei auf der Festplatte nicht dasselbe. Bei einem Ratespiel zeigt das Programm Aufforderungen wie

```
Geben Sie eine ganze Zahl zwischen 1 und 10 ein
an. Das Programm steht mit der Person, die an der Tastatur sitzt,
in einem Dialog. Einen solchen Dialog gibt es in Listing 8.2 nicht.
Das Programm Gehaltsabrechnung liest Zeichen von der
Festplatte ein und fordert niemandem auf, etwas zu tun oder
einzugeben.
```

Der größte Teil dieses Kapitels beschäftigt sich mit der Wiederverwendung von Code. [Listing 8.2](#) hat mit einer Idee zu tun, die nicht unmittelbar die Wiederverwendung von Code betrifft. Es liest, anders als die Beispiele in den vorherigen Kapiteln, Daten aus einer gespeicherten Datei aus. Deshalb unternehme ich in den nächsten Abschnitten einen kleinen Ausflug in die Welt der Dateien auf Festplatten.

Daten in einer Datei ablegen

Der Code in [Listing 8.2](#) funktioniert erst dann, wenn es in einer Datei `Mitarbeiterdaten` gibt. Die entsprechende Datei heißt in [Listing 8.2](#) `MitarbeiterInfo.txt`. Ich habe deshalb, bevor ich den Code in [Listing 8.2](#) ausführe, erst einmal eine kleine Textdatei erstellt. [Abbildung 8.3](#) zeigt, wie diese Datei aussehen muss. (Denken Sie daran, dass Java beim Einlesen von Daten des Typs `double` eine landesspezifische Schreibweise des Dezimalzeichens verlangt!)


```
Barry Burd
Vorstand
5000,00
Harriet Ritter
Manager
7000,00
Ihr Name
Aufsichtsrat
10000,00
|
```

[Abbildung 8.3](#): Die Datei *MitarbeiterInfo.txt*



Damit [Listing 8.2](#) nicht zu kompliziert wird, müssen Sie unbedingt auch die letzte Zeile in [Abbildung 8.3](#) mit [Enter] abschließen. (Wenn Sie sich [Abbildung 8.3](#) noch einmal ansehen, fällt Ihnen vielleicht auf, dass der Cursor am Anfang einer neuen, leeren Zeile steht.) Wenn Sie vergessen, den Text durch Drücken von [Enter] zu beenden, stürzt der Code in [Listing 8.2](#) ab, wenn Sie versuchen, ihn auszuführen.



Die Separatoren zum Gruppieren von Zahlen unterscheiden sich von Land zu Land. Die Datei, die in [Abbildung 8.3](#) gezeigt wird, funktioniert nur bei einem »Komma-orientierten« Ländercode, in dem 5000,00 »fünftausend« bedeutet. Diese Datei funktioniert nicht in einem Land, in dem der Punkt das Dezimalzeichen ist (wie zum Beispiel in den USA) und 5000.00 »fünftausend« bedeutet. Wenn Sie in der Datei das falsche Dezimalzeichen verwenden, erhalten Sie beim Ausführen des Codes mit ziemlicher Sicherheit eine Fehlermeldung vom Type `InputMismatchException`. Ändern Sie in diesem Fall das Dezimalzeichen so, dass es zu den landesspezifischen Einstellungen Ihres Computers passt, und alles sollte fehlerfrei laufen.

Code kopieren und einfügen

In so gut wie jeder Programmiersprache ist das Auslesen von Daten aus Dateien ziemlich trickreich. Sie benötigen zusätzliche Codezeilen, damit der Computer weiß, was er zu tun hat.

Manchmal können Sie diese Zeilen aus dem Code anderer Entwickler kopieren. Sie können zum Beispiel der Struktur von [Listing 8.2](#) folgen:

```
/*
 * Die Struktur von Listing 8.2
 */
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
class EinKlassenName {
    public static void main(String args[])
                                throws IOException
{
    Scanner scannerName =
        new Scanner(new File("NameEinerDatei"));
    //Hierhin kommt Code
    scannerName.nextInt();
    scannerName.nextDouble();
    scannerName.next();
    scannerName.nextLine();
    //Hierhin kommt Code
    scannerName.close();
}
}
```

Sie möchten Daten lesen, die sich in einer Datei befinden. Beginnen Sie damit, dass Sie sich vorstellen, Tastatureingaben auszulesen. Bringen Sie in Ihrem Programm die üblichen Codes für `Scanner` und `next` unter. Fügen Sie dann noch ein paar Elemente aus der Struktur von [Listing 8.2](#) hinzu:

- ✓ Fügen Sie zwei neue Importdeklarationen hinzu – eine für `java.io.File` und eine für `java.io.IOException`.
- ✓ Schreiben Sie in den Kopf Ihrer Methode `throws IOException`.
- ✓ Schreiben Sie in den Aufruf von `new Scanner` den Text `new File("")`.
- ✓ Nehmen Sie eine Datei, die sich auf Ihrer Festplatte befindet. Setzen Sie den Namen dieser Datei zwischen die Anführungszeichen.

✓ Merken Sie sich den Namen Ihres Scanners. Verwenden Sie diesen Namen in den Aufrufen von `next`, `nextInt`, `nextDouble` und so weiter.

✓ Verwenden Sie den Namen Ihres Scanners beim Aufruf von `close`.

Gelegentlich kann Sie das Kopieren und Einfügen von Code auch in Schwierigkeiten bringen. Vielleicht schreiben Sie ein Programm, das nicht in die einfache Struktur von [Listing 8.2](#) passt. Sie müssen die Struktur umbauen. Um das aber sauber hinzubekommen, müssen Sie verstehen, welche Idee sich hinter der Struktur verbirgt.

Und hier rettet Sie der nächste Abschnitt. Er behandelt einige dieser Ideen.

Eine Datei auslesen

In den früheren Kapiteln lesen Programme Zeichen, die über die Tastatur des Computers eingegeben werden. Diese Programme verwenden Komponenten wie `Scanner`, `System.in` und `nextDouble` – Komponenten, die in Javas API definiert worden sind. Das Programm *Gehaltsabrechnung* in [Listing 8.2](#) bereichert die Geschichte um etwas vollkommen Neues: Statt Zeichen von der Tastatur liest das Programm Zeichen aus der Datei `Mitarbeiter Info.txt` aus. Diese Datei befindet sich auf der Festplatte Ihres Computers.

Um Zeichen aus Dateien zu lesen, verwenden Sie einige der Dinge, die auch für das Auslesen von Tastatureingaben verwendet werden. Sie setzen `Scanner`, `nextDouble` und andere tolle Sachen ein. Aber leider gibt es auch ein paar zusätzliche Hürden, die Sie überwinden müssen. Hier eine Liste davon:

✓ **Sie benötigen ein Objekt `new File`.** Um es genauer auszudrücken: Sie benötigen eine neue Instanz der API-Klasse `File`. Sie erhalten diese Instanz über Code wie `new File("MitarbeiterInfo.txt")`

Bei dem Objekt in Anführungszeichen handelt es sich um den Namen einer Datei – einer Datei auf der Festplatte Ihres Computers. Die Datei enthält Zeichen wie die in [Abbildung 8.3](#).

An dieser Stelle macht die Terminologie aus Mücken Elefanten. Klar, ich verwende die Ausdrücke `new File Objekt` und `new File Instanz`, aber sie machen nichts, als mit `new File("MitarbeiterInfo.txt")` eine Datei auf der Festplatte zu bezeichnen. Nachdem Sie `new File("MitarbeiterInfo.txt")` mit

Scanner diskScanner =

new Scanner(new File("MitarbeiterInfo.txt"));

in `new Scanner` untergebracht haben, können Sie alles vergessen, was mit `new File` zu tun hat. Ab jetzt bezeichnet `diskScanner MitarbeiterInfo.txt` eine Datei auf der Festplatte Ihres Computers.



Sie können in [Listing 8.2](#) das Erstellen des Objekts `new File` mit dem Erstellen des Objekts `new Mitarbeiter` weiter hinten im Code oder mit `new Konto` in den Beispielen von [Kapitel 7](#) vergleichen. Der einzige Unterschied sieht so aus, dass die Klassen `Mitarbeiter` und `Konto` in den Beispielen dieses Buches definiert werden, während die Klasse `File` aus der Java-API stammt.



Wenn Sie über `new Scanner` eine Verbindung zu einer Datei auf der Festplatte aufbauen, dürfen Sie den Teil mit `new File` nicht vergessen. Wenn Sie `new Scanner("MitarbeiterInfo.txt")` ohne `new File` schreiben, macht das dem Compiler nichts aus. (Sie erhalten, bevor Sie den Code ausführen, keine Warnung oder Fehlermeldung.) Aber wenn Sie das Programm ablaufen lassen, erhalten Sie nichts von dem, was Sie erwarten.

- ✓ **Sie müssen auf die Klasse `File` verweisen, indem Sie deren vollständigen Namen verwenden: `java.io.File`.** Sie können dies mit einer `import`-Deklaration wie dieser erledigen:

```
Scanner diskScanner =  
    new Scanner(new java.io.File("MitarbeiterInf  
o.txt"));
```

- ✓ **Sie benötigen eine Klausel `throws IOException`.** Es kann viel schiefgehen, wenn sich Ihr Programm mit `MitarbeiterInfo.txt` verbindet. So kann zum Beispiel sein, dass diese Datei auf der Festplatte nicht vorhanden ist, oder die Datei befindet sich im falschen Verzeichnis. Um vor den Unannehmlichkeiten dieser Art gewappnet zu sein, gibt es in Java diverse Vorsichtsmaßnahmen. Java besteht darauf, dass Sie bestätigen, die Gefahren von `new Scanner` zu kennen.

Für diese Bestätigung stehen Ihnen mehrere Wege zur Verfügung, von denen der einfachste darin besteht, die Klausel `throws` zu nutzen. In [Listing 8.2](#) endet der Kopf der Methode `main` mit `throws IOException`. Indem Sie diese beiden Wörter hinzufügen, beruhigen Sie den Java-Compiler. Das ist so, als würden Sie ihm sagen: »Ich weiß, dass der Aufruf von `new Scanner` Probleme hervorrufen kann. Daran musst du mich nicht erinnern.« Ohne die Klausel `throws` erhalten Sie eine Fehlermeldung vom Typ `unreported exception`. (Wenn Sie mehr über Javas Ausnahmen wissen wollen, lesen Sie [Kapitel 12](#). Bis dahin fügen Sie dem Kopf einer jeden Methode, die `new Scanner(new File(...` aufruft, `throws IOException` hinzu.

- ✓ **Sie müssen auf die Klasse `IOException` mit ihrem vollständigen Namen verweisen – `java.io.IOException`.**

Sie erledigen dies über eine `import`-Deklaration wie der in [Listing 8.2](#). Alternativ können Sie aber auch die Klausel

throws der Methode `main` erweitern:

```
public static void main(String args[])  
    throws java.io.IOException {
```

✓ **Sie müssen den Namen des Dateiscanners an die Methode `bezahleMitarbeiter` übergeben.**

In [Kapitel 7](#) hat in [Listing 7.5](#) die Methode `zinsen` einen Parameter mit dem Namen `zinsSatz`. Immer wenn Sie die Methode `zinsen` aufrufen, übergeben Sie der Methode ein besonderes, aktuelles Stückchen Information. (Sie übergeben der Methode eine Zahl – einen Zinssatz. [Abbildung 7.7](#) stellt das entsprechende Konzept dar.)

Etwas Ähnliches geschieht in [Listing 8.2](#). Die Methode `bezahleMitarbeiter` besitzt den Parameter `aScanner`. Jedes Mal, wenn Sie die Methode `bezahleMitarbeiter` aufrufen, übergeben Sie der Methode ein besonderes, aktuelles Stückchen Information. (Sie übergeben der Methode einen Scanner – eine Referenz auf eine Datei.)

Vielleicht wundern Sie sich darüber, dass die Methode `bezahleMitarbeiter` einen Parameter haben muss. Immerhin liest sie in [Listing 8.2](#) jedes Mal dieselbe Datei aus. Warum also müssen Sie sich damit abplagen, die Methode bei jedem Aufruf darüber zu informieren, dass es die Datei `MitarbeiterInfo.txt` immer noch gibt?

Nun, es gibt Wege, um den Code in [Listing 8.2](#) umzubauen. Einige dieser Wege verzichten auf Parameter. Aber in unserem Beispiel läuft alles über zwei Methoden: die Methode `main` und die Methode `bezahleMitarbeiter`. In der Methode `main` legen Sie einen Scanner an, den Sie drei Mal verwenden – jeweils einmal in jedem Aufruf der Methode `bezahleMitarbeiter`.

Alles, was Sie in einer Methode deklarieren, ist wie etwas, das nur im Code dieser Methode bekannt ist. Deshalb ist der `diskScanner`, den Sie in der Methode `main` definieren, nicht automatisch auch in der Methode `bezahleMitarbeiter`

bekannt. Deshalb übergeben Sie `diskScanner` aus der Methode `main` an die Methode `bezahleMitarbeiter`. (Sie finden in [Kapitel 10](#) mehr über Komponenten heraus, die Sie innerhalb oder außerhalb von Methoden deklarieren.)

Wer hat die Datei verschoben?

Wenn Sie die Codebeispiele von der Website dieses Buches herunterladen, werden Sie auf Dateien wie `Mitarbeiter.java` und `Gehaltsabrechnung.java` stoßen – den Code in den Listings 8.1 und 8.2. Außerdem gibt es eine Datei mit dem Namen `MitarbeiterInfo.txt`. Und das ist auch gut so, denn wenn Java diese Datei nicht findet, läuft das gesamte Projekt nicht sauber, und Sie erhalten eine `FileNotFoundException`.

Im Allgemeinen besagt eine `FileNotFoundException`, dass eine Datei, die Ihr Programm benötigt, von diesem nicht gefunden wird. Zu diesem Fehler kann es sehr schnell kommen und dies kann ziemlich frustrierend sein. Für Sie ist die Datei `MitarbeiterInfo.txt` vorhanden, aber denken Sie daran, dass ein Computer dumm ist. Er kann nicht zwischen den Zeilen lesen, und wenn Sie nur einen kleinen Fehler machen, ist es aus. Wenn sich die Datei nicht auf der Festplatte Ihres Computers befindet oder Sie sich beim Dateinamen verschrieben haben, versagt Ihr Computer.

Manchmal wissen Sie ganz genau, dass es auf der Festplatte eine Datei `Mitarbeiter Info.txt` (oder *WasAuchImmer.xyz*) gibt. Aber wenn Sie dann Ihr Programm ausführen, erhalten Sie diese unschöne `FileNotFoundException`. Normalerweise liegt die Datei in solch einem Fall im falschen Verzeichnis. (Das ist natürlich eine Frage des Standpunktes. Eventuell ist die Datei zwar im richtigen Verzeichnis, aber Sie haben Java aufgefordert, im falschen Verzeichnis nachzusehen.) Kopieren Sie in diesem Fall die Datei in andere Festplattenverzeichnisse und lassen Sie den Code erneut ablaufen. Sie sollten sich so lange intensiv um Dateinamen und Dateiverzeichnisse kümmern, bis Sie den Fehler herausgefunden haben.

Den Dateinamen um den Verzeichnisnamen erweitern

Sie können in Ihrem Java-Code genau angeben, wo sich eine Datei befindet. Code wie `new File("C:UsersbburdworkspaceListing08-01-02MitarbeiterInfo.txt")` sieht zwar hässlich aus, funktioniert aber.

Beachten Sie im vorherigen Absatz den doppelten umgekehrten Schrägstrich (der umgekehrte Schrägstrich wird auch *Backslash* genannt). Wenn Sie Windows benutzen, sind Sie eigentlich daran gewöhnt, `C:Usersbburdworkspace ...` mit einem einfachen Backslash zu schreiben. Aber unter Java besitzt der einfache umgekehrte Schrägstrich (Backslash) eine eigene Bedeutung. (So sagt zum Beispiel in [Listing 7.7](#) aus, dass zu einer neuen Zeile gegangen wird.) Merken Sie sich deshalb, dass Sie in Java einen doppelten Backslash verwenden, wenn Sie in einem von Anführungszeichen eingeschlossenen String einen einfachen Backslash nutzen wollen.



Für Macintosh- und Linux-Benutzer ist es sicherlich angenehm zu erfahren, dass das Zeichen, das Sie zum Trennen von Pfadangaben verwenden (/), in einem Java-String keine reservierte Bedeutung hat. Auf einem Mac ist der Code `new`

```
File("/Users/bburd/workspace/Listing08-01-02/MitarbeiterInfo.txt")
```

 so normal wie

regelmäßiges Atmen. Aber Mac- und Linux-Benutzer sollten sich nicht zu schnell als etwas Besseres betrachten. Zeilen wie `new File("/Users/bburd/workspace ...` funktionieren auch unter Windows. Unter Windows können Sie sowohl einen Schrägstrich (/) als auch den umgekehrten Schrägstrich (\) verwenden, um Angaben zu Pfaden auf der Festplatte zu machen. Ich könnte auf der Windows-

Befehlszeile sogar `cd c:/users/bburd` eingeben, um zu meinem Home-Verzeichnis zu gelangen.



Wenn Sie wissen, wo Ihr Java-Programm Dateien sucht, können Sie den Weg dorthin genau vorgeben. Unterstellen wir einmal, dass der Code in [Listing 8.2](#) die Datei `MitarbeiterInfo.txt` normalerweise im Projektordner `08-01` sucht. Öffnen Sie als Experiment dieses Verzeichnis und legen Sie ein neues Unterverzeichnis an, das den Namen `datenDateien` bekommt. Verschieben Sie dann die Datei `MitarbeiterInfo.txt` in das neue Verzeichnis. Nun müssen Sie noch die entsprechende Zeile im Code in `new File("datenDateienMitarbeiterInfo.txt")` ändern.

Eine Zeile nach der anderen lesen

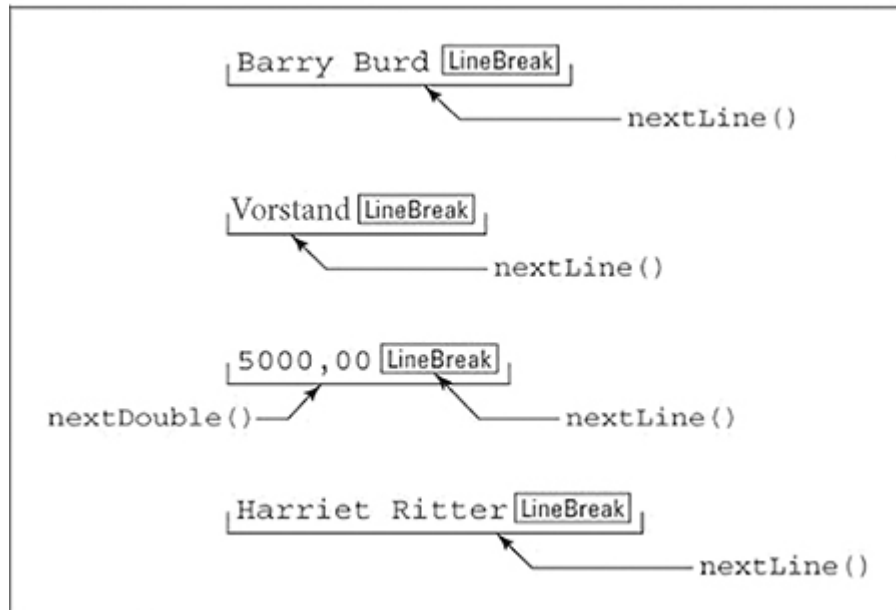
In [Listing 8.2](#) zeigt die Methode `bezahleMitarbeiter` einige nützliche Tricks, um Daten auszulesen. So verfügt zum Beispiel jeder Scanner, den Sie erstellen, über eine Methode `nextLine`. (Vielleicht möchten Sie diese Methode nicht verwenden, aber Sie sollten auf jeden Fall wissen, dass es sie gibt.) Wenn Sie die Methode `nextLine` eines Scanners aufrufen, schnappt sich die Methode alles bis zum Ende der aktuellen Textzeile. In [Listing 8.2](#) ist ein Aufruf von `nextLine` in der Lage, eine ganze Zeile der Datei `MitarbeiterInfo.txt` auszulesen. (In einem anderen Programm könnte der Aufruf von `nextLine` einer Methode dazu führen, dass alles, was ein Benutzer bis zum Drücken der Taste `[Enter]` eingibt, ausgelesen wird.)

Beachten Sie, wie vorsichtig ich mich ausdrücke: `nextLine` liest alles bis zum Ende der aktuellen Zeile. Unglücklicherweise bedeutet »bis zum Ende der aktuellen Zeile« nicht immer das, was Sie normalerweise darunter verstehen. Gemischte Aufrufe von `nextInt`, `nextDouble` und `nextLine` können leicht in einem Chaos enden. Sie müssen aufpassen, was Sie machen,

und Sie müssen die Ausgabe Ihres Programms sorgfältig beobachten.

Wichtig ist, dass Sie peinlich genau auf die Zeilenumbrüche einer Datendatei achten. Stellen Sie sich einen Zeilenumbruch als ein zusätzliches Zeichen vor, das zwischen zwei Zeilen steckt. Und stellen Sie sich nun vor, dass der Aufruf von `nextLine` bedeutet, dass alles bis einschließlich dieses Zeilenumbruchzeichens gelesen wird.

Werfen Sie einen Blick auf [Abbildung 8.4](#).



[Abbildung 8.4](#): `nextDouble` und `nextLine` aufrufen

- ✓ Wenn ein Aufruf von `nextLine` `Barry Burd[LineBreak]` liest, liest das darauf folgende `nextLine` die berufliche Position `Vorstand[LineBreak]` (wobei `LineBreak` das »Zeichen« für den Zeilenumbruch ist).
- ✓ Wenn ein Aufruf von `nextDouble` die Zahl `5000,00` liest, liest das darauf folgende `nextLine` das `[LineBreak]`, das unmittelbar auf die Zahl folgt. (Eigentlich ist das auch alles, was `nextLine` liest – ein `[LineBreak]` und sonst nichts.)

✓ Wenn ein Aufruf von `nextLine` das `[LineBreak]` hinter der Zahl `5000,00` liest, liest der dann folgende Aufruf von `nextLine` `Harriet Ritter[LineBreak]`.

Das bedeutet nun, dass Sie nach dem Lesen der Zahl `5000,00` **zwei** Aufrufe von `nextLine` haben müssen, um an den Namen `Harriet Ritter` zu gelangen. Normalerweise vergesse ich den ersten dieser beiden Aufrufe.



Schauen Sie sich noch einmal die Datei in [Abbildung 8.3](#) an. Damit der Code dieses Abschnitts läuft, benötigen Sie hinter dem letzten `10000,00` ein *Line Break* (einen Zeilenumbruch). Ist dies nicht der Fall, sorgt der letzte Aufruf von `nextLine` dafür, dass Ihr Programm abstürzt. Die Fehlermeldung, die Sie daraufhin erhalten, lautet `NoSuchElementException: No line found`.



Ich bin immer wieder über die Macken erstaunt, die ich in den Scanner-Methoden von Programmiersprachen finde. So liest zum Beispiel das erste `nextLine` in der Datei aus [Abbildung 8.3](#) `Barry Burd[LineBreak]`, liefert aber nur `Barry Burd` ohne den Zeilenumbruch (`[LineBreak]`) an den laufenden Code zurück. `nextLine` sucht also das Zeichen für den Zeilenumbruch, um es schnellstens wieder zu verlieren.



Wenn Sie die Sache mit `nextDouble` und `nextLine` irritiert, sollten Sie die Schuld dafür nicht bei Java suchen. Aufrufe von gemischten Dateneingaben sind in jeder Programmiersprache eine heikle Sache. Besonders unangenehm ist dabei, dass jede Programmiersprache anders an das Problem herangeht. Wenn Sie etwas über `nextLine` in Java wissen, hilft Ihnen das dabei, dieses Thema in C++ oder Visual Basic zu verstehen, wobei hier aber die entsprechenden sprachspezifischen Einzelheiten nicht geliefert werden. Klar, alles zu verstehen, kostet Zeit

und Mühe, aber da alle Programmierer reich und berühmt werden, lohnt sich der Einsatz allemal.

Die Verbindung zu einer Datei auf der Festplatte beenden

Für den normalen Computerbenutzer sind eine Tastatur und eine Datei auf der Festplatte des Computers zwei grundsätzlich andere Dinge. Aber Dateien und Eingaben über die Tastatur haben viel gemeinsam. So schreibt einer der Grundsätze von Betriebssystemen vor, dass bei einer Programmierung die Unterschiede bei Eingabemöglichkeiten so weit wie möglich verschwimmen müssen. Sie sollten als Java-Programmierer Eingaben über die Tastatur und Dateien möglichst gleich behandeln. Das ist der Grund dafür, dass [Listing 8.2](#) einen Aufruf von `diskScanner.close()` enthält.

Wenn Sie ein Java-Programm ablaufen lassen, führen Sie normalerweise die Befehle der Methode `main` aus. Dabei beginnen Sie mit dem ersten Befehl im Körper der Methode und hören mit dem letzten Befehl im Körper der Methode auf. Auf diesem Weg können Sie Umleitungen einbauen, `else`-Teile überspringen und in die Körper von Methoden eintauchen, aber grundsätzlich beenden Sie die Ausführung mit dem letzten Befehl in der Methode `main`. Aus diesem Grund steht in [Listing 8.2](#) der Aufruf von `close` am Ende des Körpers der Methode `main`. Wenn Sie den Code in diesem Listing ausführen, beenden Sie als Letztes die Verbindung zur Datei auf der Festplatte.

Unterklassen definieren (was es bedeutet, ein Vollzeit- oder ein Teilzeitmitarbeiter zu sein)

Vor einem Jahr hat Ihr Unternehmen zehn Millionen Euro für Software ausgegeben, die in der Datei `Mitarbeiter.class`

geliefert wurde. Die Mitarbeiter von Burd Brain Consulting (der Firma, die das Programm erstellt hat) möchten nicht, dass Sie die Interna des Programms zu Gesicht bekommen (weil man Angst hat, dass Sie deren Ideen stehlen). Das heißt, dass Sie den Quellcode der Software nicht besitzen (Ihnen die Datei `Mitarbeiter.java` nicht zur Verfügung steht). Sie können den Bytecode in der Datei `Mitarbeiter.class` ablaufen lassen. Sie können auch die Dokumentation lesen, die als `Mitarbeiter.html` mitgeliefert wurde. Aber Sie sind nicht in der Lage, sich die Befehle anzusehen, und Sie können den Code des Programms nicht ändern.

Seitdem ist Ihr Unternehmen gewachsen, und anders als früher gibt es jetzt zwei Arten von Mitarbeitern: Vollzeit- und Teilzeitmitarbeiter. Jeder in Vollzeit arbeitende Mitarbeiter erhält wöchentlich ein festes Gehalt. (Wenn ein solcher Mitarbeiter nachts und an den Wochenenden arbeitet, erhält er für den damit verbundenen, nicht unerheblichen Aufwand ein herzliches Dankeschön.) In Teilzeit arbeitende Mitarbeiter werden auf Stundenbasis bezahlt. Das Unternehmen zieht vom Gehalt der in Vollzeit arbeitenden Mitarbeiter etwas für spätere besondere Bonuszahlungen und Sozialleistungen ab. Teilzeitmitarbeiter erhalten diese Bonuszahlungen nicht.

Die Frage ist nun, wie die Software, die das Unternehmen im letzten Jahr gekauft hat, mit dessen Wachstum mithalten kann. Sie haben in ein großartiges Programm investiert, das Mitarbeiter und deren Gehaltsabrechnungen verarbeitet, das aber zwischen Vollzeit- und Teilzeitmitarbeitern nicht unterscheiden kann. Sie haben nun mehrere Möglichkeiten:

- ✓ Fragen Sie die zwölfjährige Tochter Ihres Nachbarn, die mehr über Programmierung weiß als jeder aus Ihrer Firma. Lassen Sie dieses hochnäsige kleine Balg die Mitarbeiter-Software auseinandernehmen, neu schreiben und dann mit allen gewünschten Erweiterungen und Änderungen an Sie zurückgeben.

Das war nur ein Gedankenblitz, denn so etwas können Sie nicht machen. Ganz egal, wie schlaue dieses Kind auch sein mag, so wird es wohl doch von der Komplexität der Aufgabe überfordert sein. Wenn Sie die Software zurückbekommen, ist sie voller Fehler und Ungereimtheiten. Und, was noch wichtiger ist, Sie dürfen dem Kind die Datei nicht aushändigen. Sie besitzen nur die Datei `Mitarbeiter.class`, die sich mit einem Texteditor nicht lesen oder bearbeiten lässt (siehe [Kapitel 2](#)). Aber abgesehen davon, haben gerade Ihre Kinder die Nachbarskinder verdroschen, und Sie wollen sich doch jetzt nicht die Blöße geben, ausgerechnet diese Kinder um Hilfe zu bitten.

- ✓ Verschrotten Sie die zehn Millionen Euro teure Mitarbeiter-Software. Finden Sie im Unternehmen jemanden, der das Programm vollkommen neu schreibt.

Mit anderen Worten: Sagen Sie »Tschüss« zu Freizeit und Ihrem Geld.

- ✓ Schreiben Sie für die Mitarbeiter-Software ein neues Frontend. Das heißt, dass Sie Code schreiben, der vorbereitende Arbeiten bei Vollzeitmitarbeitern übernimmt und die Ergebnisse der Verarbeitung an Ihre zehn Millionen teure Software übergibt. Machen Sie dasselbe für Teilzeitmitarbeiter.

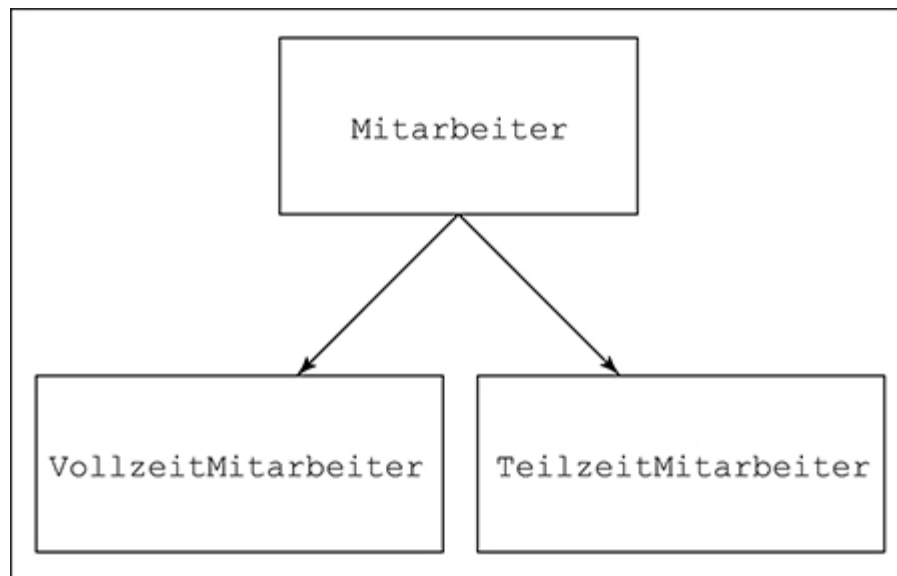
Diese Überlegungen können entweder erfolgreich sein oder in einem Desaster enden. Sind Sie sicher, dass die vorhandene Mitarbeiter-Software genügend Eintrittspunkte hat, die es Ihrer Frontendsoftware erlaubt, die Daten, die Teilergebnisse, die die neue Vorverarbeitung liefert, an die teure Mitarbeiter-Software zu senden? Denken Sie daran, dass Ihr Plan die vorhandene Software als einen großen monolithischen Block behandelt, was ziemlich umständlich werden kann. Es ist nicht leicht, die Arbeit zwischen Ihrem Frontend-Code und dem vorhandenen Mitarbeiter-Programm aufzuteilen. Und wenn Sie der *Black Box* Mitarbeiter-Programm Ebene um Ebene hinzufügen, können Sie leicht bei einem ineffizienten System enden.

✓ Rufen Sie Burd Brain Consulting an, die Firma, die Ihnen die Mitarbeiter-Software verkauft hat. Sagen Sie Dr. Burd, dass Sie die nächste Version seiner Software benötigen, um zwischen Vollzeit- und Teilzeitmitarbeitern unterscheiden zu können.

»Kein Problem,« sagt Dr. Burd. »Alles wird bis zum nächsten Abrechnungsquartal fertig sein.« Am Abend tätigt Dr. Burd dann einen diskreten Anruf bei seinem Nachbarn ...

✓ Erstellen Sie zwei neue Java-Klassen mit dem Namen `VollzeitMitarbeiter` und `TeilzeitMitarbeiter`. Beide neuen Klassen sollen zwar die Funktionalität der vorhandenen teuren Klasse `Mitarbeiter` erweitern, aber jede dieser neuen Klassen soll jeweils eigene Funktionen definieren, die speziell für eine bestimmte Art von Mitarbeitern gedacht sind.

Das ist es! [Abbildung 8.5](#) stellt die Struktur dar, die Sie anlegen wollen.



[Abbildung 8.5](#): Der Stammbaum der Klasse *Mitarbeiter*

Unterklassen erstellen

Ich definiere in [Listing 8.1](#) eine Klasse `Mitarbeiter`. Das, was ich dort angelegt habe, kann ich verwenden und die Definition

erweitern, um neue, speziellere Klassen zu erstellen. In [Listing 8.3](#) lege ich fest, wie die neue Klasse `VollzeitMitarbeiter` aussehen soll.

```
public class VollzeitMitarbeiter extends Mitarbeiter
{
    private double wochenLohn;
    private double sozialAbzug;
    public void setWochenLohn(double wochenLohnIn) {
        wochenLohn = wochenLohnIn;
    }
    public double getWochenLohn() {
        return wochenLohn;
    }
    public void setSozialAbzug(double sozialAbzugIn)
    {
        sozialAbzug = sozialAbzugIn;
    }
    public double getSozialAbzug() {
        return sozialAbzug;
    }
    public double berechneZahlung() {
        return wochenLohn - sozialAbzug;
    }
}
```

[Listing 8.3](#): Was macht einen Vollzeitmitarbeiter aus?

Sie können in [Listing 8.3](#) sehen, dass jede Instanz der Klasse `VollzeitMitarbeiter` zwei Felder aufweist: `wochenLohn` und `sozialAbzug`. Aber sind dies wirklich die einzigen Felder, die jede Instanz von `VollzeitMitarbeiter` hat? Nein. Die erste Zeile von [Listing 8.3](#) sagt aus, dass die Klasse `VollzeitMitarbeiter` die vorhandene Klasse `Mitarbeiter` erweitert (`extends`). Dies bedeutet, dass jede Instanz von `VollzeitMitarbeiter` zusätzlich zu ihren beiden Feldern `wochenLohn` und `sozialAbzug` auch über die beiden Felder `name` und `funktion` verfügt. Diese beiden Felder stammen aus der Definition der Klasse `Mitarbeiter`, die Sie [Listing 8.1](#) entnehmen können.

Das magische Wort lautet in [Listing 8.3](#) `extends` (deutsch *erweitert*). Wenn eine Klasse eine vorhandene Klasse erweitert, erbt die erweiternde Klasse automatisch Funktionen, die in der vorhandenen Klasse definiert werden. Aus diesem Grund *erbt* die Klasse `VollzeitMitarbeiter` die Felder `name` und `funktion`. Außerdem erbt sie alle Methoden, die in der Klasse `Mitarbeiter` **deklariert** werden: `setName`, `getName`, `setFunktion`, `getFunktion` und `bezahleGehalt`. Die Klasse `VollzeitMitarbeiter` ist eine *Unterklasse* der Klasse `Mitarbeiter`. Dies bedeutet, dass die Klasse `Mitarbeiter` die *übergeordnete Klasse* oder *Superklasse* der Klasse `VollzeitMitarbeiter` ist. Und um die Begrifflichkeit voll zu machen, können Sie auch noch in Ausdrücken reden, die mit verwandtschaftlichen Beziehungen zu tun haben. Die Klasse `VollzeitMitarbeiter` ist das *Kind* der Klasse `Mitarbeiter`, die umgekehrt die *Elternklasse* der Klasse `VollzeitMitarbeiter` ist.

[Listing 8.4](#) erweckt den Anschein, als wenn sein Code die Klasse `VollzeitMitarbeiter` definieren würde – was aber nicht ganz stimmt.

```
import static java.lang.System.out;
public class VollzeitMitarbeiter {
    private String name;
    private String funktion;
    private double wochenLohn;
    private double sozialAbzug;
    public void setName(String nameIn) {
        name = nameIn;
    }
    public String getName() {
        return name;
    }
    public void setFunktion(String funktionIn) {
        funktion = funktionIn;
    }
    public String getFunktion() {
        return funktion;
    }
    public void setWochenLohn(double wochenLohnIn) {
```

```

        wochenLohn = wochenLohnIn;
    }
    public double getWochenLohn() {
        return wochenLohn;
    }
    public void setSozialAbzug(double sozialAbzugIn)
{
        sozialAbzug = sozialAbzugIn;
    }
    public double getSozialAbzug() {
        return sozialAbzug;
    }
    public double berechneZahlung() {
        return wochenLohn - sozialAbzug;
    }
    public void zahleGehalt(double betrag) {
        out.printf("Zahlen Sie an %s ", name);
        out.printf("(%) ***", funktion);
        out.printf("%.2f Euron", betrag);
    }
}

```

[Listing 8.4](#): Nichts als Täuschung (aber informativ)



Warum wird [Listing 8.4](#) als Täuschung bezeichnet? Nun, der größte Unterschied zwischen [Listing 8.4](#) und der vererbenden Situation der [Listings 8.1](#) und [8.3](#) ist dieser: Eine Kindklasse ist nicht in der Lage, direkt auf die privaten Felder ihrer Elternklasse zu verweisen. Damit die Kindklasse etwas mit den privaten Feldern ihrer Elternklasse anfangen kann, muss sie die Zugriffsmethoden der Elternklasse aufrufen. So wäre zum Beispiel in [Listing 8.3](#) der Aufruf `setName("Rufus")` zulässig, was aber nicht für den Code `name = "Rufus"` gilt. Falls Sie alles glauben, was Sie in [Listing 8.4](#) lesen, könnten Sie auf die irrige Idee kommen, dass `VollzeitMitarbeiter` in der Lage wäre, mit `name = "Rufus"` umgehen.



Die Datei `Mitarbeiter.java` muss sich nicht auf Ihrer Festplatte befinden, damit Sie Code schreiben können, der

die Klasse `Mitarbeiter` erweitert. Sie benötigen nur die Datei `Mitarbeiter.class`.

Das Anlegen von Unterklassen kann zur Gewohnheit werden

Nachdem Sie sich daran gewöhnt haben, Klassen zu erweitern, können Sie sich daranmachen, auch eine Klasse `TeilzeitMitarbeiter` zu erstellen (siehe Listing 8.5).

```
public class TeilzeitMitarbeiter extends Mitarbeiter
{
    private double stundenSatz;
    public void setStundenSatz (double sSatzIn) {
        stundenSatz = sSatzIn;
    }
    public double getStundenSatz() {
        return stundenSatz;
    }
    public double berechneZahlung(int stunden) {
        return stundenSatz * stunden;
    }
}
```

[Listing 8.5](#): Was macht einen Teilzeitmitarbeiter aus?

Mit Unterklassen arbeiten

Der vorherige Abschnitt erzählt Ihnen eine Geschichte darüber, wie eine Unterklasse erstellt wird. Es ist eine schöne Geschichte, aber sie ist unvollständig. Zwar ist das Erstellen von Unterklassen eine gute Sache, aber Sie haben erst dann etwas davon, wenn Sie den Code schreiben, der es Ihnen möglich macht, diese Art von Klassen zu nutzen. Aus diesem Grund untersuchen Sie in diesem Abschnitt Code, der Unterklassen verwendet.

Jetzt ist für Sie die Zeit gekommen, sich selbst einzustufen. Sind Sie eine Person vom Typ W, eine vom Typ P oder eine vom Typ T? (Da ich der Autor dieses Buches bin, nehme ich mir die Freiheit heraus, Persönlichkeitstypen zu erfinden. Ich gehe sogar so weit zu behaupten, dass ich in aller Öffentlichkeit über jemanden sagen kann: »Das ist eine Person vom Typ T.«)

- ✓ **Eine Person vom Typ W** will das Wesentliche sehen. (Der Buchstabe W steht für *Wesentliches*.) »Zeige mir ein Programm, das die Grundlagen in ihrer reinen, ursprünglichen Form darlegt«, sagt die Person vom Typ W. Sie macht sich nichts aus Schnickschnack. Wenn Sie eine Person vom Typ W sind, erwarten Sie ein Programm, das die Unterklassen `VollzeitMitarbeiter` und `TeilzeitMitarbeiter` verwendet und Sie dann in Ruhe lässt, damit Sie Ihre Arbeit erledigen können.
- ✓ **Eine Person vom Typ P** erwartet praxisnahe Anwendungen. (Der Buchstabe P steht für *Praxis*.) Personen dieses Typs haben Schwierigkeiten, sich Ideen abstrakt vorzustellen. »Zeige mir ein Programm, das den Nutzen der Unterklassen `VollzeitMitarbeiter` und `TeilzeitMitarbeiter` aufzeigt«, sagt die Person vom Typ P. »Ich kann mit deinen blöden Abstraktionen nichts anfangen. Ich brauche Beispiele aus dem täglichen Leben, und das sofort.«
- ✓ **Eine Person vom Typ T** wird von etwas inspiriert, über das ich kurz in [Kapitel 7](#) schreibe. Eine Person dieses Typs will den Code in den Unterklassen `VollzeitMitarbeiter` und `TeilzeitMitarbeiter` *testen*. Den Code testen, bedeutet, den Code auf Herz und Nieren zu prüfen – die Genauigkeit der Ausgabe bei Standardeingaben, bei unerwarteten Eingaben und sogar bei unrealistischen Eingaben zu kontrollieren. Und für diese Tests bevorzugt die Typ-T-Person standardisierte, einfach nachzuvollziehende Regeln, damit ihre Programmierer die Testergebnisse schneller verstehen können. Die Person vom Typ T erstellt *JUnit*-Tests, die die Unterklassen `VollzeitMitarbeiter` und `TeilzeitMitarbeiter` verwenden.

[Listing 8.6](#) ist für die Typ-W-Personen gedacht. Es ist schlank und einfach und eignet sich als Bettlektüre.



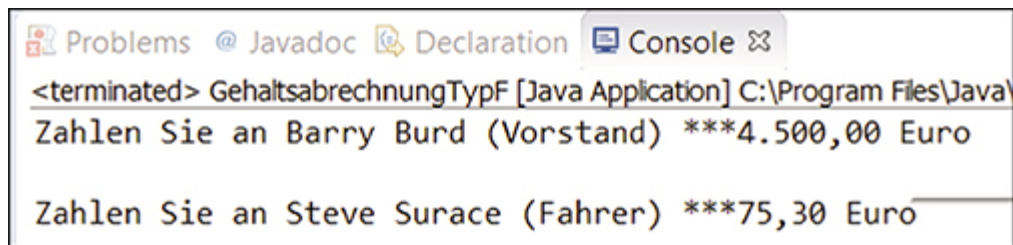
Wenn Sie vom Typ P oder T sind, sollten Sie die amerikanische Website dieses Buches besuchen

(www.allmycode.com/JavaForDummies). Die Site enthält Beispiele, um auch diese Typen zu befriedigen. Beachten Sie aber, dass Sie die Beispiele dieser Site am besten immer mit den originalen (nicht an die deutsche Sprache angepassten) Beispieldaten des Autors verwenden.

[Listing 8.6](#) zeigt Ihnen ein skelettartiges Programm, das die Unterklassen `Vollzeit Mitarbeiter` und `TeilzeitMitarbeiter` verwendet. [Abbildung 8.6](#) gibt wieder, was das Programm ausgibt, wenn es ausgeführt wird.

```
public class GehaltsabrechnungTypF {
    public static void main(String args[]) {
        VollzeitMitarbeiter vzMa = new VollzeitMitarbeite
r();
        vzMa.setName("Barry Burd");
        vzMa.setFunktion("Vorstand");
        vzMa.setWochenLohn(5000.00);
        vzMa.setSozialAbzug(500.00);
        vzMa.zahleGehalt(vzMa.berechneZahlung());
        System.out.println();
        TeilzeitMitarbeiter tzMa = new TeilzeitMitarbeite
r();
        tzMa.setName("Steve Surace");
        tzMa.setFunktion("Fahrer");
        tzMa.setStundenSatz(7.53);
        tzMa.zahleGehalt
            (tzMa.berechneZahlung(10));
    }
}
```

[Listing 8.6](#): Unterklassen einsetzen



```
<terminated> GehaltsabrechnungTypF [Java Application] C:\Program Files\Java\
Zahlen Sie an Barry Burd (Vorstand) ***4.500,00 Euro

Zahlen Sie an Steve Surace (Fahrer) ***75,30 Euro
```

[Abbildung 8.6](#): Das gibt das Programm in [Listing 8.6](#) aus.

Um [Listing 8.6](#) zu verstehen, müssen Sie drei Klassen im Auge behalten: `Mitarbeiter`, `VollzeitMitarbeiter` und `TeilzeitMitarbeiter`. (Wenn Sie einen Blick auf diese Klassen werfen wollen, schauen Sie sich die [Listings 8.1](#), [8.3](#) und [8.5](#) an.)

Die erste Hälfte von [Listing 8.6](#) hat mit einem `VollzeitMitarbeiter` zu tun. Beachten Sie, dass die Variable `vzMa` sehr viele Methoden nutzen kann. So können Sie beispielsweise `vzMa.setWochenLohn` aufrufen, weil `vzMa` vom Typ `VollzeitMitarbeiter` ist. Sie können aber auch `vzMa.setName` aufrufen, weil die Klasse `VollzeitMitarbeiter` die Klasse `Mitarbeiter` erweitert.

Da `zahleGehalt` in der Klasse `Mitarbeiter` deklariert wird, können Sie auch `vzMa.berechneZahlung` aufrufen. Außerdem können Sie `vzMa.berechneZahlung` aufrufen, weil sich die Methode `berechneZahlung` in der Klasse `VollzeitMitarbeiter` befindet.

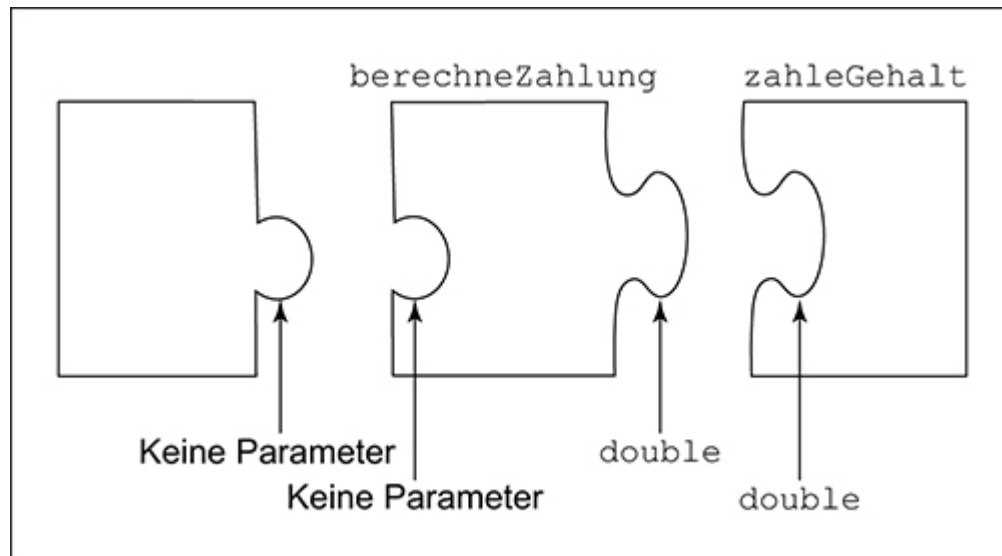
Typen passend gestalten

Schauen Sie sich noch einmal die erste Hälfte von [Listing 8.6](#) an. Achten Sie dort ganz besonders auf den letzten Befehl, der dafür sorgt, dass das Gehalt des Vollzeitmitarbeiters bezahlt wird. Der Befehl besteht aus einer Aneinanderreihung von Werten und deren Typen. Sie erkennen dies am besten dadurch, dass Sie den Befehl von innen nach außen lesen.

- ✓ Die Methode `vzMa.berechneZahlung` wird mit einer leeren Parameterliste aufgerufen ([Listing 8.6](#)). Das ist auch gut so, denn die Methode `berechneZahlung` enthält keine Parameter ([Listing 8.3](#)).
- ✓ Die Methode `berechneZahlung` gibt einen Wert vom Typ `double` zurück (siehe noch einmal [Listing 8.3](#)).
- ✓ Der `double`-Wert, den `vzMa.berechneZahlung` zurückgibt, wird an die Methode `vzMa.zahleGehalt` übergeben ([Listing 8.6](#)). Und so soll das auch sein, denn der Parameter der

Methode `bezahleGehalt` ist vom Typ `double` (siehe [Listing 8.1](#)).

[Abbildung 8.7](#) bildet diese Struktur fantasievoll nach.



[Abbildung 8.7](#): Parameter passend gestalten



Übergeben Sie an Methoden immer nur Werte, deren Typen zu den Parameterlisten passen.

Die zweite Hälfte der Geschichte

In der zweiten Hälfte von [Listing 8.6](#) erzeugt der Code ein Objekt vom Typ `TeilzeitMitarbeiter`. Eine Variable vom Typ `TeilzeitMitarbeiter` kann teilweise dieselben Dinge wie die Variable `VollzeitMitarbeiter` erledigen. Was die Klasse `TeilzeitMitarbeiter` nicht kennt, sind die Methoden `setWochenLohn` und `setSozialAbzug`. Stattdessen enthält die Klasse `TeilzeitMitarbeiter` die Methode `setStundenSatz` (siehe [Listing 8.5](#)). Deshalb handelt es sich bei der vorletzten Zeile in [Listing 8.6](#) um einen Aufruf dieser Methode.

Die letzte Zeile von [Listing 8.6](#) ist gleichzeitig auch die interessanteste. In dieser Zeile übergibt der Code die Zahl 10 (die Zahl der Arbeitsstunden) an die Methode `berechneZahlung`.

Vergleichen Sie dies mit dem früheren Aufruf von `berechneZahlung` – dem Aufruf für den in Vollzeit arbeitenden Mitarbeiter in der ersten Hälfte von [Listing 8.6](#). Es gibt für jede der beiden Unterklassen `VollzeitMitarbeiter` und `TeilzeitMitarbeiter` eine eigene Methode `berechneZahlung`. Und jede dieser beiden Methoden verfügt über eine eigene Parameterliste:

✓ Die Methode `berechneZahlung` der Klasse `VollzeitMitarbeiter` weist keine Parameter auf ([Listing 8.3](#)).

✓ Die Methode `berechneZahlung` der Klasse `TeilzeitMitarbeiter` enthält einen Parameter ([Listing 8.5](#)).

Das war zu erwarten. Das Berechnen eines Betrags für einen Teilzeitmitarbeiter ist nun einmal nicht dasselbe wie das Berechnen des Gehalts eines Vollzeitmitarbeiters. Das Gehalt eines Teilzeitmitarbeiters ändert sich von Woche zu Woche, während ein in Vollzeit arbeitender Mitarbeiter immer dasselbe Gehalt bekommt. Deshalb enthalten die beiden Klassen `VollzeitMitarbeiter` und `Teilzeitmitarbeiter` jeweils eine Methode `berechneZahlung`, die dann aber eigene Arbeitsweisen haben.

Methoden überschreiben (Zahlungen für einige Mitarbeiter ändern)

Als ob Sie nicht damit gerechnet hätten: Irgendein Schwachkopf in der Personalabteilung bietet einem der Teilzeitmitarbeiter einen hundertprozentigen Aufschlag auf den Stundenlohn bei Überstunden an. Das spricht sich herum und nun wollen alle Teilzeitmitarbeiter für Überstunden die doppelte Bezahlung haben. Wenn sich das durchsetzt, enden Sie im Armenhaus. Also setzen

Sie sich hin und verfassen ein Schreiben an alle Teilzeitmitarbeiter, in dem Sie erklären, warum eine Lohnerhöhung nicht in ihrem Sinne wäre.

In der Zwischenzeit haben Sie es mit zwei Arten von Teilzeitmitarbeitern zu tun – mit denen, deren Überstunden mit dem doppelten Satz vergütet werden, und mit denen, bei denen sich nichts ändert. Sie müssen also die Software für die Gehaltsabrechnung ändern. Welche Möglichkeiten haben Sie?

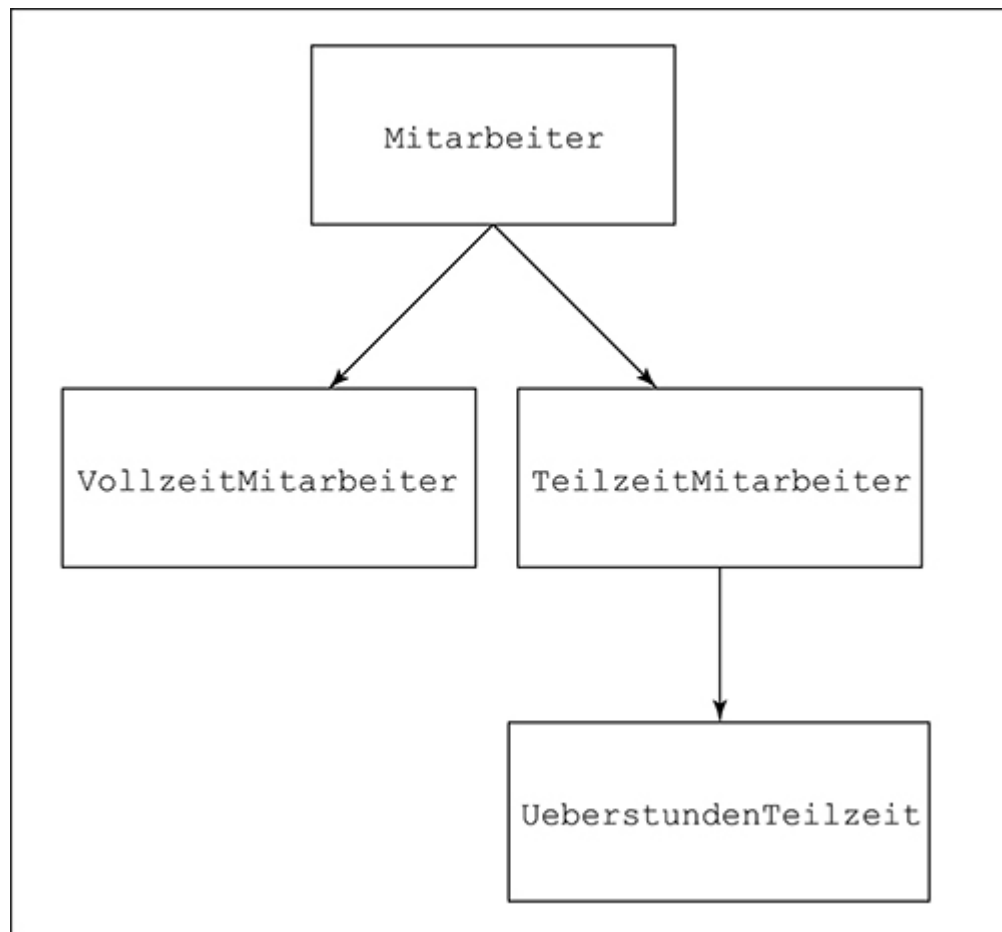
- ✓ Sie können sich tief in den Code der Klasse `TeilzeitMitarbeiter` stürzen, Änderungen vornehmen und das Beste hoffen. (Kein wirklich guter Plan.)
- ✓ Sie können dem Tipp des vorherigen Abschnitts folgen und eine Unterklasse der Klasse `TeilzeitMitarbeiter` erstellen. Sie könnten nun natürlich festhalten, dass die vorhandene Klasse `TeilzeitMitarbeiter` bereits über eine Methode `berechneZahlung` verfügt, und sich fragen, welche trickreichen Wege Sie einschlagen müssen, um diese Methode zu umgehen, wenn es um Überstunden geht, für die das Doppelte gezahlt wird.

Jetzt ist wieder einmal der Zeitpunkt gekommen, an dem Sie allen Göttern dafür danken sollten, dass Sie in Java objektorientiert programmieren. Bei dieser Art der Programmierung erstellen Sie eine Unterklasse, die die Funktionen ihrer Elternklasse überschreibt. [Listing 8.7](#) enthält solch eine Unterklasse.

```
public class UeberstundenTeilzeit
    extends TeilzeitMitarbeiter {
    @Override
    public double berechneZahlung(int stunden) {
        if(stunden <= 40) {
            return getStundenSatz() * stunden;
        } else {
            return getStundenSatz() * 40 +
                getStundenSatz() * 2
                * (stunden - 40);
        }
    }
}
```

[Listing 8.7](#): Eine weitere Unterklasse

[Abbildung 8.8](#) zeigt, in welcher Beziehung der Code in [Listing 8.7](#) und der restliche Code in diesem Kapitel zueinander stehen. So ist `UeberstundenTeilzeit` die Unterklasse einer Unterklasse. In der objektorientierten Programmierung ist solch eine Kette nichts Ungewöhnliches. Verglichen mit der Wirklichkeit ist unsere Kette sogar ziemlich kurz.

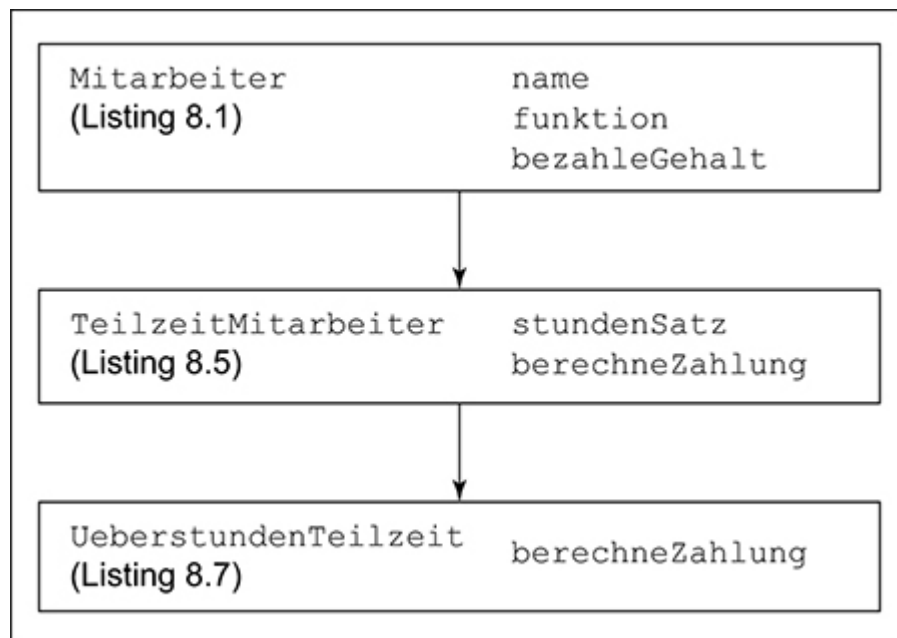


[Abbildung 8.8](#): Ein Klassenbaum

Die Klasse `UeberstundenTeilzeit` erweitert die Klasse `TeilzeitMitarbeiter`, ist aber wählisch, wenn es um das geht, was sie von ihrer Elternklasse erbt. Da `UeberstundenTeilzeit` eine eigene Deklaration der Methode

`berechneZahlung` enthält, wird diese Methode nicht von der Klasse `TeilzeitMitarbeiter` geerbt (siehe [Abbildung 8.9](#)).

Offiziell wird das, was hier geschieht, als *Überschreiben* bezeichnet – die Klasse `UeberstundenTeilzeit` *überschreibt* die Methode `berechneZahlung` ihrer Elternklasse. Wenn Sie über die Klasse `UeberstundenTeilzeit` ein Objekt erstellen, besitzt dieses Objekt zwar `name`, `funktion`, `stundenSatz` und `zahleGehalt` der Klasse `TeilzeitMitarbeiter`, aber keine eigenständig definierte Methode `berechneZahlung`.



[Abbildung 8.9](#): Die Methode `berechneZahlung` wird nicht geerbt.

Eine Java-Annotation

Das Wort `@Override` in [Listing 8.7](#) ist ein Beispiel für eine *Annotation*. (*Annotation* wird normalerweise mit *Anmerkung* übersetzt, aber die hier beschriebenen Annotations haben mit dem deutschen Ausdruck nichts zu tun, weshalb wir den englischen Begriff beibehalten.) Eine Java-Annotation erzählt Ihrem Computer etwas über Ihren Code. So weist gerade die Annotation `@Override` in [Listing 8.7](#) den Compiler an, auf

Standardcodierungsfehler zu achten. Die Annotation sagt: »Sorge dafür, dass die Methode, die dieser Annotation direkt folgt, dieselben Dinge (denselben Namen, dieselben Parameter und so weiter) enthält wie eine der Methoden in der übergeordneten Klasse. Wenn dies nicht der Fall ist, gib eine Fehlermeldung aus.«

Falls ich also irrtümlich `public double berechneZahlung(double stunden)` statt `int stunden` wie in den [Listing 8.5](#) und [8.7](#) schreibe, informiert mich mein Compiler darüber, dass meine neue Methode `berechneZahlung` nichts von dem überschreiben kann, was in [Listing 8.5](#) steht.

Java kennt noch weitere Annotations (zum Beispiel `@Deprecated` und `@SuppressWarnings`). Ich gehe in [Kapitel 9](#) kurz auf die Annotation `@SuppressWarnings` ein.



Java-Annotations sind optional. Wenn Sie in [Listing 8.7](#) das Wort `@Override` entfernen, läuft der Code trotzdem fehlerfrei weiter. Aber diese Annotation sorgt in Ihrem Code für mehr Sicherheit. Wenn `@Override` vorhanden ist, sorgt die Überprüfung des Compilers dafür, dass Sie das tun, was Sie eigentlich auch vorhaben (nämlich das Überschreiben einer Methode der übergeordneten Klasse). Und mit einem Blick auf George Orwell muss ich doch festhalten, dass einige Annotations weniger optional sind als andere. Es gibt Annotations, auf die Sie in Ihrem Code nur dann verzichten können, wenn Sie sie aufwendig durch Java-Code ersetzen.

Methoden aus Klassen und Unterklassen verwenden

Wenn Sie wegen des Überschreibens einer Methode noch Klärungsbedarf haben, schauen Sie sich den Code in [Listing 8.8](#) an. Was dieser Code ausgibt, wenn er ausgeführt wird, zeigt [Abbildung 8.10](#).

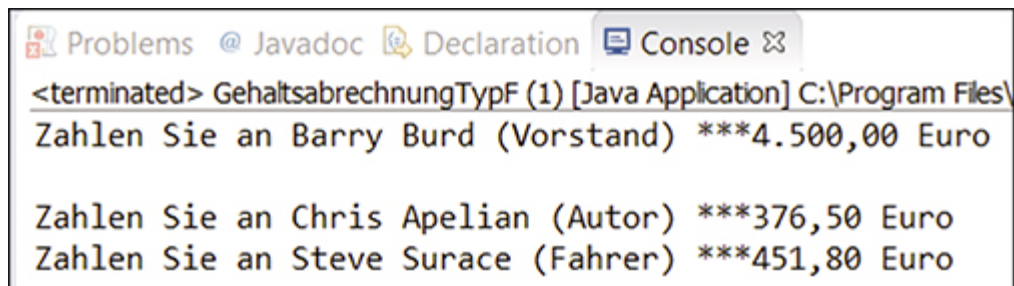
```
public class GehaltsabrechnungTypF {  
    public static void main(String args[]) {
```

```

        VollzeitMitarbeiter vzMa = new VollzeitMitarbeite
r();
        vzMa.setName("Barry Burd");
        vzMa.setFunktion("Vorstand");
        vzMa.setWochenLohn(5000.00);
        vzMa.setSozialAbzug(500.00);
        vzMa.zahleGehalt(vzMa.berechneZahlung());
        System.out.println();
        TeilzeitMitarbeiter tzMa = new TeilzeitMitarbeite
r();
        tzMa.setName("Chris Apelian");
        tzMa.setFunktion("Autor");
        tzMa.setStundenSatz(7.53);
        tzMa.zahleGehalt
                (tzMa.berechneZahlung(10));
        UeberstundenTeilzeit tzueMa = new UeberstundenTei
lzeit();
        tzueMa.setName("Steve Surace");
        tzueMa.setFunktion("Fahrer");
        tzueMa.setStundenSatz(7.53);
        tzueMa.zahleGehalt
                (tzueMa.berechneZahlung(50));
    }
}

```

[Listing 8.8](#): Den Code aus [Listing 8.7](#) testen



[Abbildung 8.10](#): Der Code aus [Listing 8.7](#) wird ausgeführt.

Der Code in [Listing 8.7](#) schreibt Überweisungen für drei Mitarbeiter. Beim ersten Mitarbeiter handelt es sich um einen Vollzeitmitarbeiter. Der zweite ist ein Teilzeitmitarbeiter, der noch keinen Wind von der neuen Regelung für die Bezahlung von

Überstunden bekommen hat, und der dritte Mitarbeiter kennt die Regelung und verlangt eine faire Bezahlung.

Durch die Unterklassen bestehen alle drei Mitarbeitertypen in [Listing 8.8](#) nebeneinander. Klar, eine Unterklasse stammt von der alten Klasse `TeilzeitMitarbeiter` ab, aber das bedeutet nicht, dass es unmöglich ist, ein Objekt dieser Klasse zu erstellen. Tatsächlich geht Java sehr intelligent mit solch einer Situation um. [Listing 8.8](#) enthält drei Aufrufe der Methode `berechneZahlung`, die jeweils eine andere Version der Methode ansprechen:

- ✓ Beim ersten Aufruf, `vzMa.berechneZahlung`, ist die Variable `vzMa` eine Instanz der Klasse `VollzeitMitarbeiter`. Aus diesem Grund handelt es sich bei der Methode, die aufgerufen wird, um die aus [Listing 8.3](#).
- ✓ Beim zweiten Aufruf, `tzMa.berechneZahlung`, ist die Variable `tzMa` eine Instanz der Klasse `TeilzeitMitarbeiter`. Aus diesem Grund handelt es sich bei der Methode, die aufgerufen wird, um die aus [Listing 8.5](#).
- ✓ Beim dritten Aufruf, `tzueMa.berechneZahlung`, ist die Variable `tzueMa` eine Instanz der Klasse `UeberstundenTeilzeit`. Aus diesem Grund handelt es sich bei der Methode, die aufgerufen wird, um die aus [Listing 8.7](#).

Dieser Code ist fantastisch. Er ist eindeutig, elegant und effizient. Mit dem ganzen Geld, das Sie nun an der Software sparen, sind Sie problemlos in der Lage, Überstunden gerecht zu bezahlen.

9

Neue Objekte entwerfen

In diesem Kapitel

- ▶ Konstruktoren definieren
 - ▶ Konstruktoren in Unterklassen verwenden
 - ▶ Standardfunktionen der Java-Konstruktor verwenden
 - ▶ Eine einfache GUI entwerfen
-

Ms. Jennie Burd
121 Schoolhouse Lane
Anywhere, Kansas

Sehr geehrte Ms. Burd,

in Beantwortung Ihres Schreibens vom 21. Juni kann ich mit voller Überzeugung behaupten, dass sich Objekte nicht spontan aus dem Nichts heraus bilden. Auch wenn ich in der Tat noch nie gesehen habe, wie ein Objekt erstellt wurde (und es in unserem Unternehmen auch niemanden gibt, der so etwas je erlebt hat), bin ich der festen Überzeugung, dass der eine oder andere Prozess für das Erzeugen dieser interessanten und nützlichen Schnickschnacks verantwortlich ist. Wir hier bei *KlassenUndObjekte.com* unterstützen die übereinstimmenden Meinungen aus Wissenschaft und Privatwirtschaft. Darüber hinaus begrüßen wir die kürzlich erfolgte Einsetzung des Sonderausschusses im Bundestag, der zweifelsfrei zu dem Schluss gekommen ist, dass das spontane Erstellen von Objekten einen Ausblick auf die aktuelle Wirtschaftslage erschwert.

Seien Sie versichert, dass ich alle notwendigen Schritte unternommen habe, um Ihre Sicherheit und Ihr Wohlergehen zu gewährleisten. Sollten Sie noch Fragen haben, zögern Sie nicht, unsere Reklamationsabteilung zu kontaktieren. Der Leiter dieser

Abteilung ist Mr. Blake Wholl. Sie erreichen ihn am besten über die Website unseres Unternehmens.

Ich möchte mich noch einmal für Ihr Interesse bedanken und hoffe, Sie auch weiterhin zu den Stammkunden von *KlassenUndObjekte.com* zählen zu dürfen.

Mit freundlichen Grüßen

Mr. Scott Brickenchicker (der in [Kapitel 4](#) nicht mehr in den Fahrstuhl ging)

Konstrukturen definieren (was es bedeutet, eine Temperatur zu sein)

Dies ist ein Befehl, der ein neues Objekt erstellt:

```
Konto meinKonto = new Konto();
```

Ich weiß, dass das funktioniert, weil es schließlich eines meiner eigenen Beispiele in [Kapitel 7](#) ist. In diesem Kapitel schreibe ich, dass Sie, »wenn der Computer `new Konto()` ausführt, ein Objekt erstellen, indem Sie den *Konstruktor* der Klasse `Konto` aufrufen«. Was heißt das?

Wenn Sie den Computer auffordern, ein neues Objekt zu erstellen, antwortet der Computer damit, dass er bestimmte Aktionen ausführt. Zuerst sucht er in seinem Arbeitsspeicher einen Platz, an dem er die Informationen über das neue Objekt ablegen kann. Wenn das Objekt Felder aufweist, enthalten diese höchstwahrscheinlich wichtige Werte. (Sie finden in [Kapitel 7](#) mehr über Felder heraus.)

Dabei stellt sich eine Frage: Wenn Sie den Computer anweisen, ein neues Objekt zu erstellen, können Sie dann steuern, was in den Feldern des Objekts gespeichert wird? Vielleicht haben Sie, wenn der Computer ein neues Objekt anlegt, eine lange Liste mit Aufgaben, die der Computer ausführen soll. So könnten Sie zum

Beispiel den Computer auffordern, die Größe aller Schaltflächen eines Fenster anzupassen, wenn er ein neues Fenster-Objekt erstellt.

Zum Erstellen neuer Objekte können alle möglichen Aufgaben gehören, aber in diesem Kapitel geht es um Konstruktoren. Ein Konstruktor weist den Computer an, alle Aufgaben zu erledigen, die zum Erstellen eines neuen Objekts gehören.

Was ist eine Temperatur?

»Guten Morgen und willkommen zu den Objekt-Nachrichten. Die Temperatur an Ihrem Wohnort beträgt angenehme 73 Grad Fahrenheit.«

Jede Temperaturangabe besteht aus zwei Komponenten: einer Zahl und einer Temperaturskala. Eine Zahl ist einfach nur ein `double`-Wert wie 32.0 oder 70.52. Aber was ist eine Temperaturskala? Handelt es sich dabei um eine Zeichenfolge (einen String) wie "Fahrenheit" oder "Celsius"? Nicht wirklich, weil Strings nicht unbedingt Temperaturskalen sein müssen. Es gibt keine Temperaturskala "Quelploof", und ein Programm, das *73 Grad Quelploof* anzeigt, ist kein wirklich gutes Programm. Wie also können Sie die Temperaturskalen auf die paar Skalen beschränken, die normalerweise verwendet werden? Eine Möglichkeit besteht darin, Javas `enum`-Typ einzusetzen.

Was ist eine Temperaturskala? (Der Java-Typ »enum«)

Java stellt viele Wege zur Verfügung, um Dinge in Gruppen zusammenzufassen. In [Kapitel 11](#) können Sie Komponenten in einem Array oder einer Sammlung gruppieren. In diesem Kapitel fassen Sie Komponenten in einem `enum`-Typ zusammen.

(Natürlich können Sie nichts gruppieren, solange Sie nicht in der Lage sind, `enum` auszusprechen. Sagen Sie einfach *eh-nuhm*, und Sie sind dabei. Das Wort stammt vom englischen *Enumeration* ab, was auf Deutsch *Aufzählung* bedeutet.)

Es ist nicht einfach, komplizierte `enum`-Typen zu erstellen, aber ein einfacher `enum`-Typ lässt sich mit ein paar Wörtern schreiben, die in geschweiften Klammern stehen. [Listing 9.1](#) definiert solch einen `enum`-Typ, der den Namen `TemperaturSkala` bekommt.

```
public enum TempScale {  
    CELSIUS, FAHRENHEIT, KELVIN, RANKINE,  
    NEWTON, DELISLE, RÉAUMUR, RÖMER, LEIDEN  
};
```

[Listing 9.1](#): Der `enum`-Typ `TempScale`

Wenn Sie einen `enum`-Typ definieren, geschehen zwei interessante Dinge:

✓ **Sie erstellen Werte.**

Wie 13 und 151 Werte vom Typ `int` sind, sind `CELSIUS` und `FAHRENHEIT` `TempScale`-Werte.

✓ **Sie können Variablen erstellen, die diese Werte referenzieren.**

Ich deklariere in [Listing 9.2](#) die Felder `number` (für die Grad-Angaben) und `scale` (für die Aufnahme des entsprechenden Skalenwertes). Und während `double zahl` eine Variable mit dem Namen `zahl` deklariert, die vom Typ `double` ist, sorgt `TempScale scale` für die Variable `scale`, die vom Typ `TempScale` ist.

Wenn etwas vom Typ `TempScale` ist, bedeutet dies, dass es die Werte `CELSIUS`, `FAHRENHEIT`, `KELVIN` und so weiter annehmen kann. Ich habe in [Listing 9.2](#) der Variablen `scale` den Wert `FAHRENHEIT` (beziehungsweise genauer `TempScale.FAHRENHEIT`) gegeben.



Ein `enum`-Typ ist eine getarnte Java-Klasse. Aus diesem Grund entspricht [Listing 9.1](#) auch einer `.java`-Datei, die nichts als die Deklaration des `enum`-Typs `TempScale` enthält. Die Deklaration eines `enum`-Typs gehört also immer

in eine eigene Datei. Für [Listing 9.1](#) heißt dies, dass das Listing in der Datei `TempScale.java` zu finden ist.

Also gut, was ist denn nun eine Temperatur?

Jede Temperaturangabe besteht aus zwei Komponenten: einer Zahl und einer Temperaturskala. Der Code in [Listing 9.2](#) zeigt dies in allen Einzelheiten.

```
public class Temperature {
    private double number;
    private TempScale scale;
    public Temperature() {
        number = 0.0;
        scale = TempScale.FAHRENHEIT;
    }
    public Temperature(double number) {
        this.number = number;
        scale = TempScale.FAHRENHEIT;
    }
    public Temperature(TempScale scale) {
        number = 0.0;
        this.scale = scale;
    }
    public Temperature(double number, TempScale scale
) {
        this.number = number;
        this.scale = scale;
    }
    public void setNumber(double number) {
        this.number = number;
    }
    public double getNumber() {
        return number;
    }
    public void setScale(TempScale scale) {
        this.scale = scale;
    }
    public TempScale getScale() {
        return scale;
    }
}
```

[Listing 9.2](#): Die Klasse *Temperature*

Der Code in [Listing 9.2](#) enthält die üblichen Setter- und Getter-Methoden (Zugriffsmethoden für die Felder `number` and `scale`). Einzelheiten zu dieser Art von Methoden können Sie in [Kapitel 7](#) nachlesen.

Außerdem gibt es in [Listing 9.2](#) vier wie Methoden aussehende Elemente, die alle den Namen `Temperature` tragen, damit ihr Name mit dem der Klasse übereinstimmt. Keines dieser methodenähnlichen Elemente besitzt einen `return`-Typ – noch nicht einmal `void`, das eigentlich ein aussteigender `return`-Typ ist.

Jedes dieser methodenähnlichen Elemente wird *Konstruktor* genannt. Ein Konstruktor kann mit einer Methode verglichen werden, wobei er aber einem ganz besonderen Zweck dient: Er soll neue Objekte erstellen.



Immer wenn der Computer ein neues Objekt erstellt, führt er die Anweisungen in einem Konstruktor aus.



Sie können in der jeweils ersten Zeile von [Listing 9.1](#) und [9.2](#) auf das Wort `public` verzichten. Aber wenn Sie das tun, kann es dazu kommen, dass andere Java-Programme die Funktionen nicht nutzen können, die Sie in `enum TempScale` und in der Klasse `Temperature` definiert haben. (Machen Sie sich keine Gedanken über die Programme dieses Kapitels. Sie sind sowohl mit als auch ohne `public` lauffähig und nutzen problemlos den Code der [Listing 9.1](#) und [9.2](#). Sie finden in [Kapitel 13](#) heraus, welche Programme öffentliche (`public`) oder nicht öffentliche Klassen nutzen können.) Aber wenn Sie in der ersten Zeile von [Listing 9.1](#) das Wort `public` verwenden, *muss* dieses Listing in einer eigenen Datei mit dem Namen `TempScale.java` gespeichert werden. Wenn Sie das Wort `public` in der ersten Zeile von [Listing 9.2](#) verwenden, *muss*

dieses Listing in der Datei `Temperature.java` gespeichert werden. In beiden Fällen muss der Dateiname hundertprozentig mit dem Namen des darin enthaltenen Objekts übereinstimmen. (Sie können in [Kapitel 7](#) eine Einführung in das Schlüsselwort `public` nachlesen.)

Was Sie mit einer Temperatur anfangen können

[Listing 9.3](#) gibt den Ideen Gestalt, die ich im vorherigen Abschnitt beschreibe. Sie rufen in [Listing 9.3](#) die Konstruktoren auf, die in [Listing 9.2](#) deklariert werden. [Abbildung 9.1](#) zeigt, was geschieht, wenn Sie den Code ablaufen lassen.

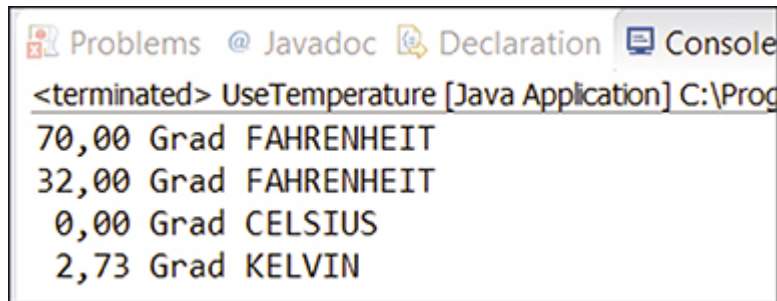
```
import static java.lang.System.out;
public class UseTemperature {
    public static void main(String args[]) {
        final String format = "%5.2f Grad %sn";
        Temperature temp = new Temperature();
        temp.setNumber(70.0);
        temp.setScale(TempScale.FAHRENHEIT);
        out.printf(format, temp.getNumber(),
                    temp.getScale());

        temp = new Temperature(32.0);
        out.printf(format, temp.getNumber(),
                    temp.getScale());

        temp = new Temperature(TempScale.CELSIUS);
        out.printf(format, temp.getNumber(),
                    temp.getScale());

        temp = new Temperature(2.73, TempScale.KELVIN)
;
        out.printf(format, temp.getNumber(),
                    temp.getScale());
    }
}
```

[Listing 9.3](#): Die Klasse *Temperature* wird verwendet.



```
<terminated> UseTemperature [Java Application] C:\Prog
70,00 Grad FAHRENHEIT
32,00 Grad FAHRENHEIT
0,00 Grad CELSIUS
2,73 Grad KELVIN
```

Abbildung 9.1: Der Code aus [Listing 9.3](#) wird ausgeführt.

In [Listing 9.3](#) ruft jede Anweisung der Art

```
temp = new Temperature(bla,bla,bla);
```

einen der Konstruktoren in [Listing 9.2](#) auf. Wenn nun der Code in [Listing 9.3](#) läuft, erstellt er vier Instanzen der Klasse `Temperature`. Jede dieser Instanzen wird durch das Aufrufen eines anderen Konstruktors erstellt.

In [Listing 9.3](#) enthält der Aufruf des letzten Konstruktors zwei Parameter: `2.73` und `TempScale.KELVIN`. Dies ist keine Besonderheit beim Aufrufen von Konstruktoren. Der Aufruf einer Methode oder der Aufruf eines Konstruktors kann einige Parameter enthalten, die durch Kommata voneinander getrennt werden. Eine andere Bezeichnung für »einige Parameter« ist *Parameterliste*.

Die einzige Regel, der Sie folgen müssen, ist die, dass die Parameter im Aufruf mit den Parametern in der Deklaration übereinstimmen müssen. So enthält, wie bereits erwähnt, in [Listing 9.3](#) der vierte und damit letzte der Konstruktoreaufrufe

```
new Temperature(2.73, TempScale.KELVIN)
```

zwei Parameter, von denen der erste vom Typ `double` und der zweite vom Typ `TempScale` ist. Java akzeptiert diesen Aufruf eines Konstruktors, weil es in [Listing 9.3](#) eine passende Deklaration gibt. Hier enthält der Kopf

```
public Temperature(double number, TempScale scale)
```

zwei Parameter, von denen der erste vom Typ `double` und der zweite vom Typ `TempScale` ist. Wenn in [Listing 9.3](#) der Aufruf eines `Temperature`-Konstruktors in [Listing 9.2](#) keine passende

Deklaration findet, stürzt [Listing 9.3](#) ab. (Java zeigt in diesem Fall eine Fehlermeldung an, wenn Sie versuchen, den Code in [Listing 9.3](#) zu kompilieren.)

Nebenbei bemerkt, die Sache mit mehreren Parametern ist nicht neu. Schon in [Kapitel 6](#) gibt es

`keyboard.findWithinHorizon(".", 0).charAt(0)`. Hier enthält der Methodenaufruf `findWithinHorizon(".", 0)` zwei Parameter – einen String und einen `int`-Wert. Ich hatte dabei das Glück, dass die Java-API für mich die Deklaration der Methode `findWithinHorizon` übernommen hat – eine Deklaration, deren erster Parameter ein String und deren zweiter Parameter ein `int`-Wert ist.

Wie »enum«-Typen und »switch«-Befehle ausgetrickst werden können

Die [Listing 9.2](#) und [9.3](#) enthalten lange Namen wie `TempScale.FAHRENHEIT` und `TempScale.CELSIUS`. Namen wie `FAHRENHEIT` und `CELSIUS` gehören zu meinem `enum`-Typ `TempScale` (den ich in [Listing 9.1](#) definiert habe). Diese Namen haben außerhalb meines `TempScale`-Kontextes keine Bedeutung. (Und wenn Sie nun glauben, dass ich wegen »meines `TempScale`-Kontextes« eingebildet sei, löschen Sie in [Listing 9.2](#) in `TempScale.FAHRENHEIT` den `TempScale.`-Teil, und plötzlich weist Sie Java darauf hin, dass Ihr Code einen Fehler enthält.)

Java ist normalerweise sehr kleinlich, wenn es um Namen und Punkte geht. Aber als die Entwickler von Java `enum`-Typen erstellt haben, entschieden sie, dass *Enums* und `switch`-Befehle einer besonderen Behandlung bedürfen. Sie können einen `enum`-Wert verwenden, um zu entscheiden, welcher `case` in einer `switch`-Anweisung ausgeführt werden soll. In solch einem Fall verwenden Sie aber im `case`-Ausdruck nicht den Namen des `enum`-Typs. So ist zum Beispiel der folgende Java-Code richtig:

```
TempScale scale = TempScale.RANKINE;
char buchstabe;
switch (scale) {
case CELSIUS:
    buchstabe = 'C';
    break;
case KELVIN:
    buchstabe = 'K';
```

```

        break;
    case RANKINE:
    case RÉAUMUR:
    case RØMER:
        buchstabe = 'R';
        break;
    default:
        buchstabe = 'X';
        break;
}

```

Ich schreibe in der ersten Zeile des Codes `TempScale.RANKINE`, weil sich diese Zeile nicht innerhalb eines `switch`-Befehls befindet. Aber in den dann folgenden Zeilen heißt es `case CELSIUS`, `case KELVIN` und `case RANKINE` ohne `TempScale`. Tatsächlich sieht es sogar so aus, dass sich Java in Form einer lauten und gemeinen Fehlermeldung beschwert, wenn ich eine `case`-Klausel erstelle und dabei `TempScale.RANKINE` schreibe.

Eine andere Temperatur (32.0) aufrufen – eine Fallstudie

Wenn der Computer einen der Befehle `new Temperature` aus [Listing 9.3](#) ausführt, muss er entscheiden, welchen der Konstruktoren aus [Listing 9.2](#) er verwendet. Er entscheidet, indem er sich die Parameterliste anschaut – das, was hinter `new Temperature` in Klammern steht. Wenn der Computer beispielsweise

```
temp = new Temperature(32.0);
```

aus [Listing 9.3](#) ausführt, sagt er sich selbst: »Bei der in Klammern stehenden Zahl `32.0` handelt es sich um einen `double`-Wert. Einer der Konstruktoren in [Listing 9.2](#) verfügt nur über einen Parameter, der auch noch vom Typ `double` ist. Der Kopf des Konstruktors sieht so aus:

```
public Temperature(double number)
```

Daraus schließe ich, dass ich die Befehle ausführen soll, die es in diesem Konstruktor gibt.« Der Computer macht dann mit der Ausführung dieser Befehle weiter:

```

this.number = number;
scale = TempScale.FAHRENHEIT;

```


Als Ergebnis dieser Aktion erhalten Sie ein brandneues Objekt, dessen Feld `number` den Wert `32.0` und dessen Feld `scale` den Wert `FAHRENHEIT` hat:

Diese beiden Codezeilen enthalten zwei Befehle, die Werte festlegen (jeweils für `number` und `scale`). Also musste derjenige, der diesen Code programmiert hat, eine Entscheidung darüber fällen, welcher Wert für das Feld `scale` verwendet werden soll. Der eine Programmierer hätte `FAHRENHEIT` oder `CELSIUS` wählen können, während sich ein anderer in [Listing 9.1](#) für `KELVIN`, `RANKINE` oder einen anderen obskuren Skalennamen hätte entscheiden können. (Der Programmierer, der für dieses Beispiel hier verantwortlich ist, lebt in den USA in New Jersey, wo die Leute im Allgemeinen immer noch die alte Fahrenheit-Temperaturskala verwenden.)

Kümmern wir uns noch einmal um den ersten der beiden Befehle. Dieser weist dem Feld `number` des neuen Objekts einen Wert zu. Dabei verwendet der Befehl einen netten, kleinen Trick, der in vielen Konstruktoren (und Methoden, die Feldern von Objekten Werte zuweisen) auftaucht. Um diesen Trick zu verstehen, sollten Sie einen Blick auf [Listing 9.4](#) werfen. Das Listing zeigt Ihnen zwei Wege, wie ich den Konstruktor hätte schreiben können.

```
//Verwenden Sie diesen Konstruktor ...
    public Temperature(double whatever) {
        number = wasauchimmer;
        scale = TempScale.FAHRENHEIT;
    }
//... oder verwenden Sie diesen Konstruktor ...
    public Temperature(double number) {
        this.number = number;
        scale = TempScale.FAHRENHEIT;
    }
//... aber bringen Sie nicht beide Konstruktoren
// in Ihrem Code unter.
```

[Listing 9.4](#): Zwei Wege, um an dasselbe Ziel zu gelangen

[Listing 9.4](#) enthält zwei Konstruktoren. Ich verwende im ersten Konstruktor zwei Namen – `number` und `wasauchimmer`. Im

zweiten Konstruktor benötige ich keine zwei Namen. Statt mir für den Parameter des Konstruktors einen neuen Namen einfallen zu lassen, verwende ich einen schon vorhandenen Namen erneut, indem ich `this.number` schreibe.

Und das geschieht nun wirklich in [Listing 9.2](#):

- ✓ Im Befehl `this.number = number` verweist *this number* auf das Feld `number` des neuen Objekts – das Feld, das ziemlich weit oben in [Listing 9.2](#) deklariert wird (siehe [Abbildung 9.2](#)).
- ✓ Im Befehl `this.number = number` verweist das Wort *number* (hinter dem Gleichheitszeichen, ohne *this*) auf den Parameter des Konstruktors (siehe noch einmal [Abbildung 9.2](#)).

```
class Temeperature {  
    private double number;  
    private TempScale scale;  
  
    public Temperature(double number) {  
        this.number = number;  
        scale = TempScale.FAHRENHEIT;  
    }  
}
```

[Abbildung 9.2](#): Was *this number* und *number* bedeuten

Im Allgemeinen verweist `this.irgendeinName` auf ein Feld, das zu dem Objekt gehört, das den Code enthält. Im Gegensatz dazu verweist das gute alte *irgendeinName* auf den am nächsten gelegenen Ort, an dem *irgendeinName* deklariert wurde. Für die Anweisung `this.number = number` ([Listing 9.2](#)) ist dieser »nächstgelegene Ort« die Parameterliste des Konstruktors `Temperature`.

Was soll das alles?

Stellen Sie sich vor, dass Ihr Code einen Konstruktor enthält – hier geht es um den ersten der beiden Konstruktoren in [Listing 9.4](#). Dem Parameter `wasauchimmer` wird eine Zahl wie zum Beispiel `32.0` übergeben. Dann weist der erste Befehl im Körper des Konstruktors diesen Wert, `32.0`, dem Feld `number` des neuen Objekts zu. Der Code funktioniert. Aber Sie müssen sich beim Schreiben dieses Codes für einen Parameter einen neuen Namen einfallen lassen – den Namen `wasauchimmer`. Dieser Name dient nur dem Zweck, dem Feld `number` des Objekts einen Wert zu übergeben. Was für eine Verschwendung! Um zwischen dem Parameter und dem Feld `number` unterscheiden zu können, benennen Sie etwas, das nur als temporärer Speicherort für den `number`-Wert dient.

Das Vergeben von Namen ist eine Kunst, keine Wissenschaft. Ich habe schon viele Phasen der Namensgebung hinter mich gebracht. Ich brauchte für einen Parameter einen neuen Namen und entschied mich für eine andere Schreibweise des originalen Variablennamens. (Ich glaube, ich habe den Parameter `numbr` oder `nuhumber` genannt.) Ich habe auch schon probiert, die Groß-/Kleinschreibung eines Namens zu ändern, um einen Parameternamen zu bekommen. (Ich habe Parameternamen wie `Number` oder `NUMBER` verwendet.) In [Kapitel 8](#) benenne ich meine Parameter, indem ich an den entsprechenden Variablennamen die Nachsilbe *In* hänge. (So gehört der Parameter `funktionIn` zur Variablen `funktion`.) Keines dieser Schemata der Namensgebung funktioniert wirklich gut. Ich schaffe es einfach nicht, mich an die skurrilen neuen Namen zu erinnern, die ich erstellt habe. Die gute Nachricht ist, dass Sie sich diese ganzen Anstrengungen mit der Namensgebung sparen können. Sie haben die Möglichkeit, einem Parameter denselben Namen wie der Variablen zu geben. Und um dann zwischen beiden unterscheiden zu können, verwenden Sie das Java-Schlüsselwort `this`.

Einige Dinge ändern sich nie

[Kapitel 7](#) stellt die Methode `printf` vor und erklärt, dass jeder Aufruf von `printf` mit einem Formatierungsstring beginnt. Der Formatierungsstring beschreibt, wie die restlichen Parameter angezeigt werden.

In früheren Beispielen ist dieser Formatierungsstring immer ein von Anführungszeichen umgebenes Literal. So lautet zum Beispiel der `printf`-Aufruf in [Listing 7.7](#)

```
out.printf("%4.2f EUn", myInterest);
```

In [Listing 9.3](#) habe ich mit der Tradition gebrochen und den Aufruf von `printf` mit einer Variablen begonnen, die ich `format` getauft habe.

```
out.printf(format, temp.getNumber(), temp.getScale())  
;
```

Dies geht so lange in Ordnung, wie meine Variable `format` vom Typ `String` ist. Und wirklich, in [Listing 9.3](#) lautet die erste Variablendeklaration

```
final String format = "%5.2f Grad %sn";
```

Beachten Sie bei dieser Deklaration der Variablen `format` ganz besonders das Wort `final`. Dieses Java-Schlüsselwort zeigt an, dass der Wert von `format` nicht geändert werden kann. Wenn ich [Listing 9.3](#) einen weiteren Zuweisungsbefehl wie

```
format = "%6.2f (%s)n";
```

hinzufügen würde, erhielte ich vom Compiler die Fehlermeldung `cannot assign a value to final variable`, die aussagt, dass einer `final`-Variablen keine (weiteren) Werte zugewiesen werden können.

Das Schlüsselwort `final` ist in [Listing 9.3](#) nicht zwingend notwendig. Es sorgt nur für einen zusätzlichen Schutz. Wenn ich `format` mit `"%5.2f Grad %sn"` initialisiere, habe ich vor, dies immer wieder zu verwenden. Ich weiß ganz genau, dass ich nicht vorhabe, den Wert der Variablen `format` zu ändern. Aber in einem Programm, das aus 10.000 Zeilen besteht, kann ich leicht den Überblick verlieren und versuchen, der Variablen `format` irgendwo tief im Code einen neuen Wert zuzuweisen. Deshalb deklariere ich die Variable `format` als `final`, um zu verhindern, dass ich unabsichtlich den Formatierungsstring ändere.

Noch mehr Unterklassen (etwas gegen das Wetter unternehmen)

In [Kapitel 8](#) wirbele ich mit der Idee von den Unterklassen ziemlich viel Staub auf – was so auch gewollt ist. Unterklassen

sorgen dafür, dass Code wiederverwendet werden kann, und wiederverwendbarer Code ist guter Code. Mit diesem Gedanken im Hinterkopf ist es an der Zeit, eine Unterklasse der Klasse `Temperature` zu erstellen. (Ich habe die Klasse im ersten Teil dieses Kapitels entwickelt.)

Für bessere Temperaturen sorgen

Nachdem Sie den Code in [Listing 9.3](#) durchgesehen haben, sind Sie der Meinung, dass die Verantwortung für die Anzeige der Temperaturen vollständig falsch angelegt worden ist. In [Listing 9.3](#) gibt es überflüssige Wiederholungen der Zeilen, die für die Ausgabe der Temperaturen zuständig sind. Ein Programmierer hätte Ihnen in den 1970er Jahren gesagt, dass Sie diese Zeilen an einer Stelle sammeln und in eine Methode umwandeln sollten. (Dieser Programmierer hätte zwar noch nicht den Begriff *Methode* verwendet, aber das interessiert im Moment nicht.) Es ist eine gute Sache, Zeilen in Methoden zu sammeln, aber bei der heutigen objektorientierten Vorgehensweise bei der Programmierung denken Sie weiter. Warum soll nicht jedes `temperature`-Objekt selbst die Verantwortung für seine Anzeige übernehmen? Wenn Sie zum Beispiel eine Methode `display` (für *anzeigen*) entwickeln, möchten Sie diese vielleicht mit anderen teilen, die Temperaturangaben einsetzen. Packen Sie also die Methode direkt in die Deklaration eines `temperature`-Objekts. Auf diese Weise hat jeder, der Ihren Code für die Temperaturangaben verwendet, leichten Zugriff auf Ihre Methode `display`.

Spielen Sie nun noch einmal das Band mit [Kapitel 8](#) ab. »Bla, bla, bla ... wollen Sie vorhandenen Code nicht ändern ... bla, bla, bla ... zu teuer, um noch einmal ganz von vorn anzufangen ... bla, bla, bla ... erweitern Sie vorhandene Funktionen.« Letztendlich lässt sich das alles in einem Wort zusammenfassen:

Wiederverwendung

Sie entscheiden deshalb, dass eine Unterklasse der Klasse `Temperature` erstellt werden soll. Bei `Temperature` handelt es

sich um die Klasse, die in [Listing 9.2](#) definiert wird. Ihre neue Unterklasse ergänzt die Funktionen der Klasse `Temperature` durch Methoden, die Werte einheitlich und nett formatiert anzeigen. Die neue Klasse, `TemperatureNice`, wird in [Listing 9.5](#) vorgestellt.

```
import static java.lang.System.out;
public class TemperatureNice extends Temperature {
    public TemperatureNice() {
        super();
    }
    public TemperatureNice(double number) {
        super(number);
    }
    public TemperatureNice(TempScale scale) {
        super(scale);
    }
    public TemperatureNice(double number, TempScale scale)
    {
        super(number, scale);
    }
    public void display() {
        out.printf("%5.2f Grad %sn",
                    getNumber(), getScale()
);
    }
}
```

[Listing 9.5](#): Die Klasse *TemperatureNice*

Beachten Sie in [Listing 9.5](#) in der Methode `display` den Aufruf der Methoden `getTemperature` und `getScale` der Klasse `Temperature`. Warum mache ich das? Nun, jeder direkte Verweis auf die Felder `number` und `scale` im Code der Klasse `TemperatureNice` würde zu Fehlermeldungen führen. Es ist schließlich so, dass das Objekt `TemperatureNice` über eigene `number`- und `scale`-Felder verfügt. (Immerhin ist `TemperatureNice` eine Unterklasse der Klasse `Temperature`, und der Code für die Klasse `Temperature` definiert die Felder `number` und `scale`.) Da aber `number` und `scale` in der Klasse `Temperature` privat deklariert wurden, kann nur Code diese

Felder direkt nutzen, der in der Klasse `Temperature` definiert wurde.



Hinterlegen Sie im Code der Klasse `TemperatureNice` keine zusätzlichen Deklarationen der Felder `number` und `scale`. Wenn Sie so etwas machen, erstellen Sie versehentlich vier verschiedene Variablen (zwei mit dem Namen `number` und zwei mit dem Namen `scale`). Sie wollen nun einem Variablenpaar Werte zuweisen und werden damit konfrontiert, dass die Werte bei der Anzeige des anderen Variablenpaares verschwunden sind.



Wenn der Code eines Objekts einen Aufruf der objekteigenen Methoden enthält, muss dieser Aufruf nicht mit einem Punkt eingeleitet werden. So ruft zum Beispiel im letzten Befehl in [Listing 9.5](#) das Objekt seine eigenen Methoden mit `getNumber()` und `getScale()` und nicht mit `einObjektName.getNumber()` und `etwasOderWasAnderes.getScale()` auf. Wenn Sie die »Punktlosigkeit« irritiert, können Sie dies dadurch ausgleichen, dass Sie auch hier das Schlüsselwort `this` verwenden. Schreiben Sie einfach in der letzten Zeile von [Listing 9.5](#) `this.getNumber()` und `this.getScale()`.

Konstruktoren für Unterklassen

Das wirklich Neue ist in [Listing 9.5](#) die Art und Weise, wie dort Konstruktoren deklariert werden. Die Klasse `TemperatureNice` hat vier eigene Konstruktoren. Wenn Sie sich bereits mit Unterklassen und Vererbung beschäftigt haben, fragen Sie vielleicht, warum diese Deklarationen von Konstruktoren benötigt werden. Erbt denn `TemperatureNice` nicht die Konstruktoren seiner Elternklasse `Temperature`? Nein, Unterklassen erben keine Konstruktoren.



Unterklassen erben keine Konstruktoren.

Das stimmt. Unterklassen erben keine Konstruktoren. In seltenen Ausnahmefällen kann es so aussehen, als ob ein Konstruktor geerbt sei, aber bei dieser Ausnahme handelt es sich um einen Zufallstreffer und nicht um das Normale. Im Allgemeinen gilt, dass Sie beim Definieren einer Unterklasse auch die Konstruktoren neu deklarieren.

Ich beschreibe die Ausnahme (die so aussieht, als ob ein Konstruktor geerbt sei) weiter hinten in diesem Kapitel im Abschnitt *Der Standardkonstruktor*.

Deshalb enthält der Code in [Listing 9.5](#) vier Konstruktoren. Jeder dieser Konstruktoren trägt den Namen `TemperatureNice` und jeder Konstruktor hat eine eigene, eindeutige Parameterliste. Das ist wenig Neues. Richtig interessant wird es, wenn es darum geht, dass jeder Konstruktor etwas aufruft, das `super` heißt (wobei `super` ein Java-Schlüsselwort ist).

In [Listing 9.5](#) steht `super` für einen Konstruktor in der Elternklasse.

- ✓ Der Befehl `super()` ruft in [Listing 9.5](#) den parameterlosen Konstruktor `Temperature()` aus [Listing 9.2](#) auf. Dieser parameterlose Konstruktor weist dem Feld `number` den Wert `0.0` und dem Feld `scale` `TempScale.FAHRENHEIT` zu.
- ✓ Der Befehl `super(number, scale)` ruft in [Listing 9.5](#) den Konstruktor `Temperature(double number, TempScale scale)` aus [Listing 9.2](#) auf. Im Gegenzug weist der Konstruktor den Feldern `number` und `scale` Werte zu.
- ✓ Ähnliches gilt in [Listing 9.5](#) für die Befehle `super(number)` und `super(scale)`, die ebenfalls Konstruktoren aus [Listing 9.2](#) aufrufen.

Der Computer entscheidet, welcher der Konstruktoren der Klasse `Temperature` aufgerufen wird, indem er sich die Parameterliste hinter dem Wort `super` anschaut. Wenn der Computer beispielsweise


```
super(number, scale);
```

aus [Listing 9.5](#) ausführt, sagt er sich selbst: »Die Felder in den Klammern, `number` und `scale`, haben die Typen `double` beziehungsweise `TempScale`. Nur einer der Temperature-Konstruktoren in [Listing 9.2](#) verfügt über Parameter vom Typ `double` und `TempScale`. Der Kopf des Konstruktors sieht so aus:

```
public Temperature(double number, TempScale scale)
```

Da das passt, werde ich die Befehle in diesem Konstruktor ausführen.«

Alles anwenden

Ich definiere in [Listing 9.5](#) die Klasse `TemperatureNice`. Nun ist es an der Zeit, diese Klasse auch einzusetzen. [Listing 9.6](#) enthält Code, der `TemperatureNice` verwendet.

```
public class UseTemperatureNice {  
    public static void main(String args[]) {  
        TemperatureNice temp = new TemperatureNice();  
        temp.setNumber(70.0);  
        temp.setScale(TempScale.FAHRENHEIT);  
        temp.display();  
        temp = new TemperatureNice(32.0);  
        temp.display();  
        temp = new TemperatureNice(TempScale.CELSIUS)  
;  
        temp.display();  
        temp = new TemperatureNice(2.73,  
                                   TempScale.KELVIN);  
        temp.display();  
    }  
}
```

[Listing 9.6](#): Die Klasse `TemperatureNice` verwenden

Der Code in [Listing 9.6](#) ähnelt stark dem Code in [Listing 9.3](#). Die großen Unterschiede sind:

✓ [Listing 9.6](#) erstellt Instanzen der Klasse `TemperatureNice`. Das heißt, dass [Listing 9.6](#) Konstruktoren aus der Klasse

TemperatureNice und nicht aus der Klasse Temperature aufruft.

✓ [Listing 9.6](#) nutzt die Methode `display` der Klasse TemperatureNice. Deshalb ist dieser Code viel sauberer als sein Gegenstück in [Listing 9.3](#).

Wenn Sie [Listing 9.6](#) ausführen, sieht das Ergebnis genau so aus wie das, das [Listing 9.3](#) liefert – Sie erreichen die letzte Zeile nur viel eleganter. (Das Ergebnis wird weiter vorn in diesem Kapitel in [Abbildung 9.1](#) gezeigt.)

Der Standardkonstruktor

Die zentrale Botschaft des vorherigen Abschnitts lautet: Unterklassen erben keine Konstruktoren. Was bedeutet dies für die Listings in [Kapitel 8](#)? In [Listing 8.6](#) lautet ein Befehl:

```
VollzeitMitarbeiter vzMa = new VollzeitMitarbeiter();
```

Das Problem dabei ist: Der Code, der (in [Listing 8.3](#)) VollzeitMitarbeiter definiert, scheint keine Konstruktoren zu enthalten. Wie schaffen Sie es dann, in [Listing 8.6](#) den Konstruktor VollzeitMitarbeiter aufzurufen?

Das funktioniert so: Wenn Sie eine Unterklasse erstellen und in Ihren Code nicht ausdrücklich eine Konstruktorendeklaration aufnehmen, erstellt Java einen Konstruktor. Dieser wird *Standardkonstruktor* genannt. Wenn Sie die Unterklasse `public VollzeitMitarbeiter` erstellen, sieht der Standardkonstruktor so aus wie in [Listing 9.7](#).

```
public VollzeitMitarbeiter() {  
    super();  
}
```

[Listing 9.7](#): Ein Standardkonstruktor

Der Konstruktor in [Listing 9.7](#) enthält keine Parameter, und sein einziger Befehl ruft den Konstruktor der Klasse auf, die Sie erweitern. (Wehe Ihnen, wenn die Klasse, die Sie erweitern, keinen parameterlosen Konstruktor enthält.)

Sie haben gerade von Standardkonstruktoren gelesen, aber aufgepasst! Es gibt nämlich eine Sache, die Sie von Standardkonstruktoren wissen sollten: Ein Standardkonstruktor wird nicht immer erstellt. Insbesondere wenn Sie eine Unterklasse anlegen und selbst Konstruktoren anlegen, fügt Java den Unterklassen keinen Standardkonstruktor hinzu (noch erben Unterklassen Konstruktoren von ihren Elternklassen).

Kann Ihnen dieses Verhalten ein Beinchen stellen? [Listing 9.8](#) enthält eine Kopie des Codes in [Listing 8.3](#), die um einen Konstruktor erweitert worden ist. Werfen Sie einen Blick auf die geänderte Version des `VollzeitMitarbeiter`-Codes.

```
public class VollzeitMitarbeiter extends Mitarbeiter
{
    private double wochenLohn;
    private double sozialAbzug;
    public VollzeitMitarbeiter(double wochenLohn) {
        this.ochenLohn = wochenLohn;
    }
    public void setWochenLohn(double wochenLohnIn) {
        wochenLohn = wochenLohnIn;
    }
    public double getWochenLohn() {
        return wochenLohn;
    }
    public void setSozialAbzug(double sozialAbzugIn)
    {
        sozialAbzug = sozialAbzugIn;
    }
    public double getSozialAbzug() {
        return sozialAbzug;
    }
    public double berechneZahlung() {
        return wochenLohn - sozialAbzug;
    }
}
```

[Listing 9.8](#): Schauen Sie, ich besitze nun einen Konstruktor!

Wenn Sie den `VollzeitMitarbeiter`-Code in [Listing 9.8](#) verwenden, funktioniert eine Zeile wie die folgende nicht:

```
VollzeitMitarbeiter vzMa = new VollzeitMitarbeiter();
```

Wenn Sie einen `VollzeitMitarbeiter`-Konstruktor erstellen, der einen `double`-Parameter verlangt, gibt es keinen automatisch erstellten parameterlosen Standardkonstruktor mehr.

Was bedeutet das nun für Sie? Wenn Sie einen Konstruktor deklarieren, müssen Sie alle Konstrukturen deklarieren, die Sie eventuell benötigen könnten. Nehmen Sie den Konstruktor in [Listing 9.7](#) und fügen Sie ihn dem Code in [Listing 9.8](#) hinzu. Dann funktioniert auch der Aufruf von `new VollzeitMitarbeiter()` wieder.

Ein Konstruktor, der mehr kann

Hier ein Zitat, das im Original irgendwo am Anfang dieses Kapitels steht: »Wenn Sie den Computer anweisen, ein neues Objekt zu erstellen, können Sie dann steuern, was in den Feldern des Objekts gespeichert wird? Vielleicht haben Sie, wenn der Computer ein neues Objekt erstellt, eine lange Liste mit Aufgaben, die der Computer ausführen soll.« Gehen wir näher auf dieses Wenn ein.

Das Beispiel dieses Abschnitts enthält einen Konstruktor, der mehr kann, als Feldern Werte zuzuweisen. Das Beispiel besteht aus den [Listings 9.9](#) und [9.10](#). [Abbildung 9.3](#) zeigt, wie das Ergebnis aussieht, wenn dieser Code ausgeführt wird.

```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JButton;
@SuppressWarnings("serial")
public class SimpleFrame extends JFrame {
    public SimpleFrame() {
        setTitle("Nicht auf die Schaltfläche klicken!");
    };

    setLayout(new FlowLayout());
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    add(new JButton("Panik"));
    setSize(340, 100);
    setVisible(true);
}
}
```

[Listing 9.9](#): Einen Frame definieren

```
public class ShowAFrame {  
    public static void main(String args[]) {  
        new SimpleFrame();  
    }  
}
```

[Listing 9.10](#): Einen Frame anzeigen



[Abbildung 9.3](#): Geraten Sie nicht in Panik.

Der Code in [Listing 9.9](#) besteht hauptsächlich aus Aufrufen von Methoden der Java-API (Application Programming Interface; deutsch: Schnittstelle für die Anwendungsprogrammierung). Dies bedeutet für Sie, dass der Code viele Namen enthält, die Sie vielleicht (noch) nicht kennen. Als ich anfing, Java zu lernen, glaubte ich dummerweise, dass es ausreicht, diese Namen zu kennen, um Java zu können. Dem ist aber nicht so, denn diese Namen sind eigentlich nichts als eine Art Handgepäck. Das echte Java besteht daraus, wie die Sprache objektorientierte Konzepte umsetzt.

Auf jeden Fall besteht in [Listing 9.10](#) die Methode `main` aus nur einem Befehl, einem Aufruf des Konstruktors in der Klasse `SimpleFrame`. (Ein *Frame* entspricht einem Fenster, damit lässt sich *Simple Frame* grob mit *einfaches Fenster* übersetzen.) Beachten Sie, dass das Objekt, das dieser Aufruf erstellt, keiner Variablen zugewiesen wird. Das ist so auch in Ordnung, denn der Code verweist nirgendwo auf dieses Objekt.

Die Klasse `SimpleFrame` enthält nur die Deklaration eines Konstruktors. Dieser Konstruktor ist weit davon entfernt, Werte in Variablen zu setzen. Seine Aufgabe ist es, Methode um Methode aus der Java-API aufzurufen.

Alle Methoden, die über den Konstruktor in der Klasse `SimpleFrame` aufgerufen werden, stammen aus der Elternklasse `JFrame`. Diese Klasse befindet sich im `javax.swing`-Paket. Dieses Paket und das Paket `java.awt` enthalten Klassen, die Ihnen dabei helfen, Fenster, Bilder, Zeichnungen und andere Elemente auf dem Bildschirm eines Computers anzuzeigen. (Im Paket `java.awt` stehen die Buchstaben *awt* für *Abstract Windowing Toolkit*.) Wenn Sie an Klatsch und Tratsch über die Schreibweise von Java-Paketen interessiert sind, sollten Sie einen Blick auf den Kasten *Pakete und Importdeklarationen* werfen. Wenn Sie bei diesem Thema wirklich mitreden wollen, lesen Sie [Kapitel 13](#).



Das, was normalerweise als *Fenster* (*Window*) bezeichnet wird, ist in der Java-API eine Instanz der Klasse `javax.swing.JFrame`.

Pakete und Importdeklarationen

In Java gibt es eine Funktion, die es Ihnen ermöglicht, Klassen in Klassengruppen zusammenzufassen. Eine solche Zusammenfassung wird *Paket* genannt. In der Java-Welt geben Programmierer diesen Paketen normalerweise lange, mit Punkten aufgefüllte Namen. Ich habe zum Beispiel den Domännennamen `allmycode.com` registriert. Ein Paket, das ich benenne, würde dann `com.allmycode.utils.textUtils` heißen. Die Java-API ist in Wirklichkeit nichts als eine große Sammlung von Paketen. In ihr gibt es Pakete mit Namen wie `java.lang`, `java.util`, `java.awt`, `javax.swing` und so weiter.

Vielleicht gelingt es mir, mit diesen Informationen über Pakete etwas gegen die Verwirrung zu unternehmen, die beim Thema `import`-Deklaration herrscht. Eine `import`-Deklaration, die das Wort `static` nicht verwendet, muss mit dem Namen eines Pakets beginnen und mit einem der folgenden Namen enden:

✓ Dem Namen einer Klasse aus dem Paket

✓ Einem Sternchen (das auf alle Klassen im Paket verweist)

So ist zum Beispiel die Deklaration

```
import java.util.Scanner;
```

gültig, weil es sich bei `java.util` um den Namen eines Pakets in der Java-API handelt und weil `Scanner` eine Klasse im Paket `java.util` ist. Der Name `java.util.Scanner` mit all den Punkten wird als *vollqualifizierter Name* der Klasse `Scanner` bezeichnet. Der vollqualifizierte Name einer Klasse enthält den Namen des Pakets, in dem die Klasse definiert wird. Sie können alles über `java.util` und `Scanner` herausbekommen, wenn Sie sich mit Javas API-Dokumentation beschäftigen.

Hier ein weiteres Beispiel. Die Deklaration

```
import javax.swing.*;
```

ist gültig, weil `javax.swing` der Name eines Pakets in der Java-API ist und das Sternchen auf alle Klassen in diesem Paket verweist. Wenn diese `import`-Deklaration am Anfang Ihres Codes steht, können Sie für die Klassen im Paket `javax.swing` Abkürzungen verwenden – Namen wie `JFrame`, `JButton`, `JMenuBar`, `JCheckBox` und viele andere.

Hier kommt noch ein Beispiel. Eine Zeile wie

```
import javax.*; //Gar nicht gut!!!
```

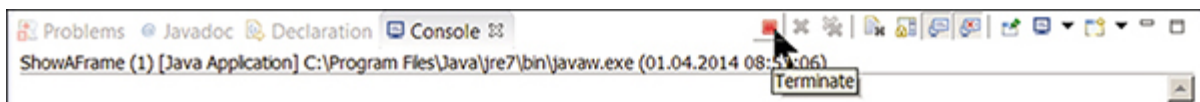
ist keine gültige `import`-Deklaration. In der Java-API gibt es kein Paket, dessen Name nur aus dem Wort `javax` besteht. Sie könnten auf die Idee kommen, dass diese Zeile Ihnen gestattet, alle Namen abzukürzen, die mit `javax` beginnen (Namen wie `javax.swing.JFrame` und `javax.sound.midi`), aber leider kann eine `import`-Deklaration damit nicht umgehen. Da es sich bei `javax` nicht um den Namen eines Pakets handelt, macht die Zeile `import javax.*` nichts anderes, als den Java-Compiler zu ärgern.

Klassen und Methoden der Java-API

Wenn Sie sich [Abbildung 9.3](#) anschauen, können Sie vielleicht bestätigen, dass eine Instanz der Klasse `SimpleFrame` eigentlich nicht viel macht. Der Frame hat nur eine Schaltfläche, und wenn Sie auf diese Schaltfläche klicken, passiert nichts. Ich habe den Frame bewusst so angelegt, damit er nicht zu kompliziert wird. Selbst dann benötigt der Code in [Listing 9.9](#)

mehrere API-Klassen und -Methoden. Die Methoden `setTitle`, `setLayout`, `setDefaultCloseOperation`, `add`, `setSize` und `setVisible` gehören alle zur Klasse `javax.swing.JFrame`. Im Code werden folgende Namen verwendet:

- ✓ `setTitle`: Der Aufruf von `setTitle` platziert Text in der Titelzeile des Frames. (Das neue `SimpleFrame`-Objekt ruft seine eigene Methode `setTitle` auf.)
- ✓ `FlowLayout`: Eine Instanz der Klasse `FlowLayout` positioniert auf dem Frame Objekte so, dass sie zentriert ausgerichtet sind. Da der Frame in [Abbildung 9.3](#) nur eine Schaltfläche enthält, wird diese im oberen Bereich des Frames zentriert. Gäbe es auf dem Frame acht Schaltflächen, würden fünf in einer ersten und die restlichen drei in einer zweiten Reihe auf dem Frame zentriert aufgereiht werden.
- ✓ `setLayout`: Der Aufruf von `setLayout` überträgt dem neuen `FlowLayout`-Objekt die Zuständigkeit für das Anordnen von Komponenten wie Schaltflächen in dem Frame. (Das neue `SimpleFrame`-Objekt ruft seine eigene Methode `setLayout` auf.)
- ✓ `setDefaultCloseOperation`: Der Aufruf von `setDefaultCloseOperation` informiert Java darüber, was zu tun ist, wenn das kleine x in der rechten oberen Ecke des Frames angeklickt wird. Ohne den Aufruf dieser Methode verschwindet der Frame zwar, aber die virtuelle Java-Maschine (JVM) läuft weiter. Wenn Sie Eclipse benutzen, müssen Sie die JVM anhalten, indem Sie in der Nähe der View CONSOLE auf das kleine rote Quadrat klicken (siehe [Abbildung 9.4](#)).



[Abbildung 9.4](#): Eclipse auffordern, die Ausführung eines Java-Programms abzuberechnen

Der Aufruf von

`setDefaultCloseOperation(EXIT_ON_CLOSE)` weist Java an, sich selbst herunterzufahren, wenn Sie in der rechten oberen Ecke des Frames auf das kleine x klicken. Alternativen zu `EXIT_ON_CLOSE` bilden `HIDE_ON_CLOSE`, `DISPOSE_ON_CLOSE` und mein Favorit `DO_NOTHING_ON_CLOSE`.

- ✓ **JButton:** Die Klasse `JButton` befindet sich im Paket `java.swing`. Einer der Konstruktoren der Klasse nimmt sich eine `String`-Instanz (wie "Panik") als Parameter. Der Aufruf dieses Konstruktors sorgt dafür, dass die `String`-Instanz auf der Oberfläche der neuen Schaltfläche erscheint.
- ✓ **add:** Das neue `SimpleFrame`-Objekt ruft seine Methode `add` auf. Der Aufruf dieser Methode platziert die Schaltfläche auf der Oberfläche des Objekts (in diesem Fall auf der Oberfläche des Frames).
- ✓ **setSize:** Der Frame wird 340 Pixel breit und 100 Pixel hoch. (Wenn Sie im Java-Paket `javax.swing` Größenangaben tätigen, kommt die Breite immer vor der Höhe.)
- ✓ **setVisible:** Wenn ein neuer Frame erstellt wird, ist er zunächst unsichtbar. Wenn dieser neue Frame aber `setVisible(true)` aufruft, erscheint er auf dem Bildschirm des Computers.

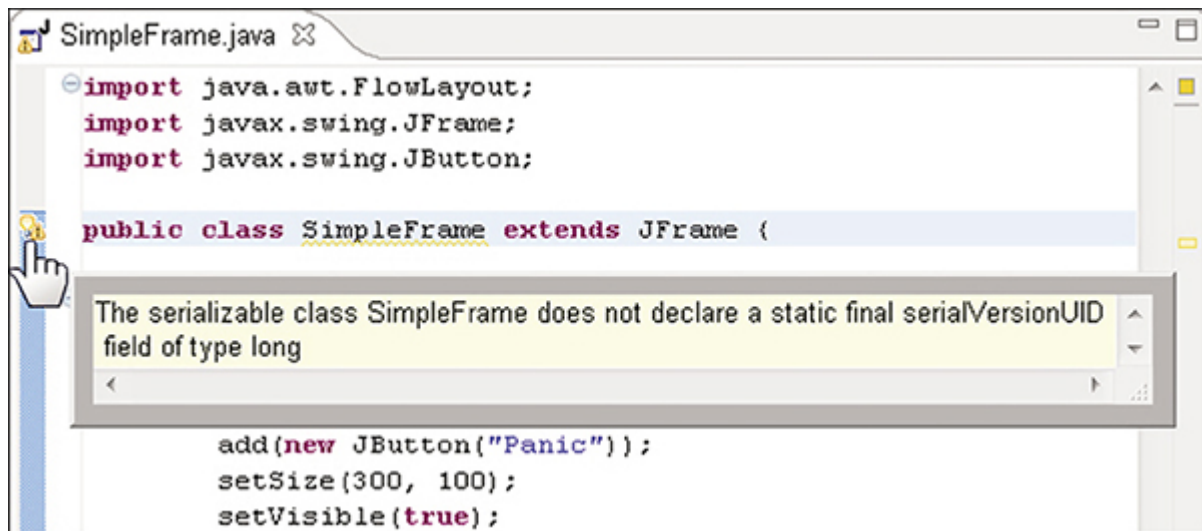
Die Annotation

»SuppressWarnings«

Kapitel 8 stellt die Annotations vor. Dabei handelt es sich um einen zusätzlichen Code, der Informationen über die Natur Ihres Programms zur Verfügung stellt. [Kapitel 8](#) beschreibt die Annotation `@Override`.

In diesem Kapitel stellt [Listing 9.9](#) eine andere Art von Annotation vor: `SuppressWarnings` (was auf Deutsch *Warnungen unterdrücken* bedeutet). Wenn Sie eine `SuppressWarnings`-

Annotation verwenden, weisen Sie Java an, Sie nicht darauf hinzuweisen, dass Ihr Programm fragwürdigen Code enthält. In [Listing 9.9](#) sagt `@SuppressWarnings("serial")` Java, dass Sie nicht darauf hingewiesen werden wollen, wenn Sie etwas nicht eingerichtet haben, das `serialVersionUID`-Feld genannt wird. Letztendlich sorgt `SuppressWarnings` dafür, dass eine Warnung wie die in [Abbildung 9.5](#) nicht angezeigt wird.



[Abbildung 9.5](#): Wenn die Annotation *SuppressWarnings* nicht vorhanden ist, warnt Java Sie davor, dass ein *serialVersionUID*-Feld fehlt.

»Was ist den bloß eine `serialVersionUID`?«, fragen Sie jetzt zu Recht. Das ist etwas, das mit dem Erweitern der Klasse `JFrame` zu tun hat – etwas, um das Sie sich wirklich nicht kümmern müssen. Wenn es kein `serialVersionUID`-Feld gibt, wird eine Warnung, keine Fehlermeldung erzeugt. Lieben Sie das gefährliche Leben! Unterdrücken Sie die Warnung (mit der Annotation in [Listing 9.9](#)) und vergessen Sie die Angelegenheit.

Teil IV

Fortgeschrittene Java-Techniken



In diesem Teil . . .

- ✓ Sie legen fest, wohin Deklarationen in einem Java-Programm gehören.
- ✓ Sie kümmern sich um Mengen (um eine Menge von Räumen, eine Menge von Verkäufen und sogar um eine Menge von Mengen).
- ✓ Sie erstellen eine App mit Fenstern und reagieren auf Mausklicks.
- ✓ Sie unterhalten sich mit der Lieblingsdatenbank.

10

Variablen und Methoden richtig platzieren

In diesem Kapitel

- ▶ Dafür sorgen, dass etwas einer ganzen Klasse gehört
 - ▶ Variablen innerhalb und außerhalb von Methoden platzieren
 - ▶ Ihre durchschnittliche Trefferquote verbessern
-

Hallo zusammen. Sie hören Radio WWW, und ich bin Ihr Moderator, Sam Burd. Und wieder einmal stehen wir am Anfang einer großartigen Baseball-Saison. Heute hat Sie Radio WWW zu einer Live-Übertragung des Spiels Hankees gegen Socks mitgenommen. Im Moment erwarte ich die Übertragung der letzten Spielminuten.

Heute Nachmittag noch sah es so aus, als ob die Socks mit den Hankees Schlitten fahren würden. Dann trafen die Hankees Ball auf Ball und machen es damit den Socks alles andere als leicht. Diese Socks! Ich bin froh, dass ich nicht in deren Haut stecken muss.

Aber was soll's. Im Verlauf des Spiels haben sich die Socks selbst aus dem Sumpf gezogen und liefern sich jetzt ein Kopf-an-Kopf-Rennen mit den Hankees. In einer Minute erhalten wir das Endergebnis und bis dahin ein paar Programmhinweise. Bleiben Sie für das große Jersey-Spiel am Radio. Und vergessen Sie nicht, auch nächste Woche wieder einzuschalten, wenn die Cleveland Gowns gegen die Bermuda Shorts spielen.

Und hier kommt das Endergebnis. Welches Team hat gewonnen? Welches Team war um eine Nasenlänge besser? Und das ist ... oh, nein! Es ist eine Krawatte.

(Anmerkung der Übersetzerin: Wundern Sie sich nicht über die Namen und die »Pointe«. *Hankees* ist die verballhornte Form des englischen Ausdrucks für *Taschentücher*, *Socks* sind *Socken*, *Jersey* bezeichnet *Trikots* oder *Pullis*, *Gowns* sind *Kleider* und *Bermuda Shorts* bleiben *Bermuda Shorts*. Diese Bezeichnungen sind beibehalten worden, weil darauf die Beispiele dieses Kapitels beruhen.)

Klassen definieren (Was es heißt, ein Baseballspieler zu sein)

Soweit es mich betrifft, hat ein Baseballspieler einen Namen und einen sogenannten Batting Average. (Ein *Batting Average* ist das Maß für die Treffsicherheit eines Spielers, der Wert, wie oft er im Durchschnitt – englisch *Average* – trifft – englisch *to bat*.) [Listing 10.1](#) wandelt meine Vorstellungen in ein Java-Programm um.

```
import java.text.DecimalFormat;
public class Player {
    private String name;
    private double average;

    public Player(String name, double average) {
        this.name=name;
        this.average=average;
    }
    public String getName() {
        return name;
    }
    public double getAverage() {
        return average;
    }

    public String getAverageString() {
        DecimalFormat decFormat = new DecimalFormat()
;
        decFormat.setMaximumIntegerDigits(0);
        decFormat.setMaximumFractionDigits(3);
```

```

        DecimalFormat.setMinimumFractionDigits(3);
        return DecimalFormat.format(average);
    }
}

```

[Listing 10.1](#): Die Klasse *Player*

Lassen Sie uns den Code in [Listing 10.1](#) in seine Einzelteile zerlegen. Glücklicherweise wird vieles von dem, was dieser Code enthält, bereits in früheren Kapiteln behandelt. Der Code definiert eine Instanz der Klasse `Player`. Er enthält:

- ✓ **Die Deklaration der Felder `name` und `average`.** Ich empfehle [Kapitel 7](#) als Bettlektüre, wenn Sie Fragen zum Deklarieren von Feldern haben.
- ✓ **Einen Konstruktor, um neue Instanzen der Klasse `Player` herzustellen.** [Kapitel 9](#) enthält eine Übersicht über Konstruktoren.
- ✓ **Zugriff auf Methoden für die Felder `name` und `average`.** Für ein kleines Schwätzchen über Zugriffsmethoden (das heißt über Setter- und Getter-Methoden) siehe [Kapitel 7](#).
- ✓ **Eine Methode, die den *Batter Average* als `String` zurückgibt.** Nette Worte über Methoden finden Sie in [Kapitel 7](#). (Habe ich nicht unheimlich viele gute Themen in [Kapitel 7](#) untergebracht?)

Ein anderer Weg, um Zahlen zu formatieren

In [Listing 10.1](#) nimmt die Methode `getAverageString` den Wert, den das Feld `average` (die durchschnittliche Trefferquote eines Spielers) hat, und sendet diesen `String`-Wert direkt an den Aufrufer der Methode zurück. Dadurch, dass `DecimalFormat` verwendet wird, das direkt aus der Java-API stammt, wird sichergestellt, dass der `String`-Wert wie eine gut formatierte Zahl und gleichzeitig wie der *Batting Average* eines Baseballspielers aussieht. Es entspricht den Aufrufen von `DecimalFormat.setMaximum...` und

`decFormat.setMinimum...`, dass der `String`-Wert links vom Dezimalzeichen keine und rechts vom Dezimalzeichen drei Ziffern enthält.

Die Java-Klasse `DecimalFormat` kann sehr praktisch sein. Um zum Beispiel die Werte 345 und -345 in einem zu einer Rechnung passenden Format anzuzeigen, können Sie den folgenden Code verwenden:

```
DecimalFormat decFormat = new DecimalFormat();
decFormat.setMinimumFractionDigits(2);
decFormat.setNegativePrefix("(");
decFormat.setNegativeSuffix(")");
System.out.println(decFormat.format(345));
System.out.println(decFormat.format(-345));
```

Bei diesem kleinen Beispiel einer `String`-Formatierung legt alles vor dem Semikolon fest, wie positive Zahlen angezeigt werden, und alles hinter dem Semikolon ist für die Ausgabe negativer Zahlen zuständig. Aufgrund dieser Formatierung werden die beiden Zahlen so angezeigt:

```
345,00
-345,00
```

Wenn Sie daran interessiert sind, weitere Tricks bei der Formatierung von Zahlen kennenzulernen, besuchen Sie die `DecimalFormat`-Seite der Dokumentation der Java-API.

Die Klasse »Player« verwenden

Die [Listings 10.2](#) und [10.3](#) enthalten Code, der die Klasse `Player` (deutsch *Spieler*) verwendet – die Klasse, die bereits in [Listing 10.1](#) definiert wird.

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
import javax.swing.JFrame;
import javax.swing.JLabel;
import java.awt.GridLayout;
@SuppressWarnings("serial")
public class TeamFrame extends JFrame {
    public TeamFrame() throws IOException {
        Player player;
```



```

        Scanner hankeesData =
            new Scanner(new File("Hankees.txt
"));
        for (int num = 1; num <= 9; num++) {
            player = new Player(hankeesData.nextLine(
),
                                hankeesData.nextDoubl
e());
            hankeesData.nextLine();

            addPlayerInfo(player);
        }
        setTitle("Die Hankees");
        setLayout(new GridLayout(9, 2, 20, 3));
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setVisible(true);
        hankeesData.close();
    }

    void addPlayerInfo(Player player) {
        add(new JLabel(" " + player.getName()));
        add(new JLabel(player.getAverageString()));
    }
}

```

[Listing 10.2](#): Die Klasse *Player* verwenden

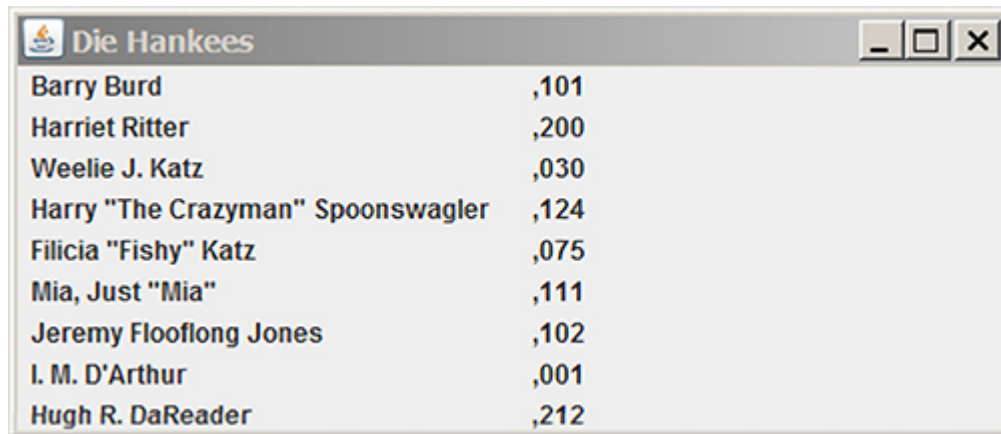
```

import java.io.IOException;
public class ShowTeamFrame {
    public static void main(String args[])
                                throws IOException {
        new TeamFrame();
    }
}

```

[Listing 10.3](#): Einen Frame anzeigen

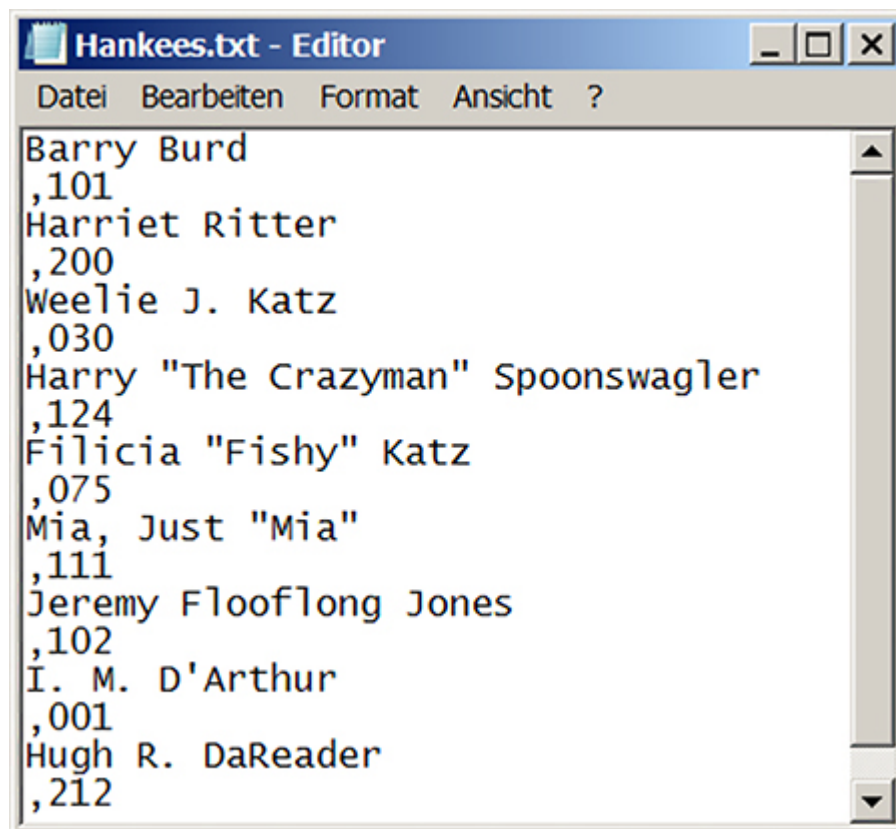
[Abbildung 10.1](#) gibt wieder, was angezeigt wird, wenn Sie den Code aus den [Listings 10.1](#) bis [10.3](#) ablaufen lassen.



Barry Burd	,101
Harriet Ritter	,200
Weelie J. Katz	,030
Harry "The Crazyman" Spoonswagler	,124
Filicia "Fishy" Katz	,075
Mia, Just "Mia"	,111
Jeremy Flooflong Jones	,102
I. M. D'Arthur	,001
Hugh R. DaReader	,212

[Abbildung 10.1](#): Würden Sie auf diese Spieler Geld setzen?

Damit Sie selbst dieses Programm ablaufen lassen können, benötigen Sie die Datei `Hankees.txt`. Diese Datei enthält die Daten Ihrer Lieblingsbaseballspieler (siehe [Abbildung 10.2](#)).



```
Barry Burd
,101
Harriet Ritter
,200
Weelie J. Katz
,030
Harry "The Crazyman" Spoonswagler
,124
Filicia "Fishy" Katz
,075
Mia, Just "Mia"
,111
Jeremy Flooflong Jones
,102
I. M. D'Arthur
,001
Hugh R. DaReader
,212
```

[Abbildung 10.2](#): Was für ein Team!

Die Datei `Hankees.txt` muss sich an einem bestimmten Ort auf der Festplatte Ihres Computers befinden. Wenn Sie Eclipse einsetzen, handelt es sich bei diesem »bestimmten Ort« um einen Projektordner in Ihrem Workspace. Wenn Sie Java aus der Befehlszeile heraus ausführen, sollte der »Ort« das Verzeichnis sein, das den Code von [Listing 10.3](#) enthält. Wenn Sie versuchen, das Beispiel dieses Abschnitts ohne die Datei `Hankees.txt` auszuführen, erhalten Sie eine unschöne `FileNotFoundException`-Meldung. Sollten Sie die Datei noch nicht heruntergeladen haben, denken Sie daran, dass sie Bestandteil der Beispieldateien dieses Buches ist, die Sie unter www.wiley-vch.de/publish/dt/books/ISBN3-527-71070-1 herunterladen können.



Damit der Code dieses Abschnitts sauber läuft, muss es hinter dem letzten , 212 in [Abbildung 10.2](#) einen Zeilenumbruch geben (siehe hierzu auch [Kapitel 8](#)).

Neunmal

Der Code in [Listing 10.2](#) ruft den Konstruktor `Player` neunmal auf. Dies bedeutet, dass der Code neun Instanzen der Klasse `Player` erstellt. Beim ersten Schleifendurchgang erstellt der Code eine Instanz mit dem Namen `Barry Burd`. Beim zweiten Schleifendurchlauf gibt der Code die `Barry Burd`-Instanz auf und erstellt eine neue Instanz mit dem Namen `Harriet Ritter`. Beim dritten Durchlauf gibt der Code die arme `Harriet Ritter` auf und erstellt die Instanz `Weelie J. Katz`. Der Code enthält immer nur eine Instanz gleichzeitig, aber er erstellt insgesamt neun Instanzen.

Jede Instanz von `Player` enthält ihre eigenen Felder `name` und `average`. Außerdem verfügt jede Instanz über einen eigenen Konstruktor `Player` und eigene Methoden `getName`, `getAverage` und `getAverageString`. Schauen Sie sich [Abbildung 10.3](#) an und denken Sie dabei an die Klasse `Player` mit ihren neun Verkörperungen.

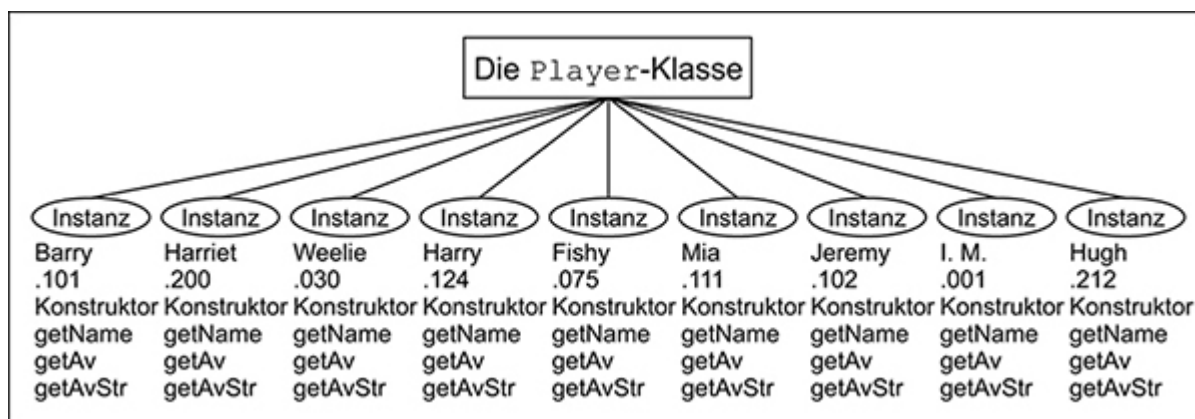


Abbildung 10.3: Eine Klasse und ihre Objekte

Eine GUI

Der Code in [Listing 10.2](#) verwendet diverse Namen aus der Java-API. Einige davon werden in [Kapitel 9](#) erklärt, andere sind jetzt an der Reihe:

✓ **JLabel:** Ein `JLabel` ist ein Objekt, das Text enthält. Eine Möglichkeit, Text in einem Frame anzuzeigen, sieht so aus, dass dem Frame eine Instanz der Klasse `JLabel` hinzugefügt wird.

In [Listing 10.2](#) wird die Methode `addPlayerInfo` neunmal aufgerufen – einmal für jeden Spieler des Teams. Jedes Mal, wenn diese Methode aufgerufen wird, fügt sie dem Frame zwei neue `JLabel`-Objekte hinzu. Der Text für die `JLabel`-Objekte stammt von der Getter-Methode eines `Player`-Objekts.

✓ **GridLayout:** Ein `GridLayout` ordnet die Dinge in gleichmäßigen Abständen in Zeilen und Spalten an. (*Grid* bedeutet auf Deutsch *Raster*, *Gitter*.) Der Konstruktor der Klasse `GridLayout` benötigt zwei Parameter: einen für die Zahl der Zeilen und einen für die Zahl der Spalten.

In [Listing 10.2](#) enthält der Aufruf des Konstruktors für `GridLayout` die Parameter `(9, 2, 20, 3)`. Deshalb werden in [Abbildung 10.1](#) neun Zeilen (eine für jeden Spieler)

und zwei Spalten (eine für den Namen und eine für den Durchschnitt) angezeigt. Der horizontale Abstand zwischen den Spalten beträgt 20 Pixel und der Abstand zwischen den einzelnen Zeilen ist drei Pixel groß.

✓ **pack:** Über `pack` legen Sie die Größe eines Frames fest. Dabei handelt es sich um die Größe, die der Frame hat, wenn er auf dem Bildschirm Ihres Computers erscheint. Der Aufruf der Methode `pack()` sorgt dafür, dass sich der Frame größentechnisch an die Objekte »anschmiegt«, die sich in ihm befinden.

In [Listing 10.2](#) haben Sie, wenn Sie den Aufruf von `pack` erreichen, schon neunmal `addPlayerInfo` aufgerufen und dem Frame 18 Labels hinzugefügt. Wenn der Computer die Methode `pack` ausführt, sucht er für jedes Label eine Größe aus, die zu dem Text passt, den Sie im Label untergebracht haben. Dann kümmert sich der Computer darum, auch für den gesamten Frame eine Größe herauszufinden, um alle 18 Labels angemessen zu positionieren.

Wenn Sie in einem Label Daten hinterlegen, haben Sie bei der Reihenfolge, in der Sie das tun, etwas Spielraum. So können Sie zum Beispiel das Layout festlegen, bevor oder nachdem Sie dem Frame Labels und andere Komponenten hinzugefügt haben. Wenn Sie `setLayout` aufrufen und dann erst Labels hinzufügen, erscheinen diese sauber angeordnet auf dem Frame. Wenn Sie umgekehrt vorgehen (erst die Labels hinzufügen und dann `setLayout` aufrufen), sorgt der Aufruf von `setLayout` dafür, dass die Labels auf wundersame Weise neu angeordnet werden. Beide Wege sind gute Vorgehensweisen.

Wenn Sie einen Frame einrichten, gibt es etwas, was Sie auf keinen Fall machen dürfen – den folgenden Ablauf missachten:

```
Dem Frame Komponenten hinzufügen dann  
pack();  
setVisible(true);
```

Wenn Sie `pack` aufrufen und versuchen, dem Frame weitere Komponenten hinzuzufügen, berücksichtigt die Methode `pack`

alles, was ganz neu ist, nicht. Wenn Sie `setVisible` aufrufen, bevor Sie Komponenten hinzugefügt oder `pack` aufgerufen haben, sieht der Benutzer den Frame in seinem Rohzustand. Und wenn Sie vergessen, die Größe des Frames festzulegen (indem Sie `pack` oder eine andere Methode aufrufen, die für eine Größenangabe zuständig ist), sieht der Frame, der dann erscheint, wie der in [Abbildung 10.4](#) aus. (Normalerweise lege ich keinen Wert darauf, das Ergebnis einer anormalen Programmausführung wie in [Abbildung 10.4](#) wiederzugeben, aber ich habe den Fehler mit `pack` so oft selbst gemacht, dass mir der mickrige Frame zu einem guten Freund geworden ist.)



[Abbildung 10.4](#): Ein unterernährter Frame

Eine Ausnahme von Methode zu Methode weiterreichen

[Kapitel 8](#) zeigt, wie Sie Daten mithilfe einer Datei eingeben können, und in diesem Zusammenhang fällt auch der Begriff *Ausnahme* oder *Exception*. Wenn Sie mit einer Datei auf der Festplatte liebäugeln, müssen Sie immer damit rechnen, dass es zu einer `IOException` kommt. Das ist die Botschaft in [Kapitel 8](#), und das ist auch der Grund dafür, dass es im Konstruktor von [Listing 10.2](#) die Klausel `throws IOException` gibt.

Wie sieht das nun aber in [Listing 10.3](#) mit der Methode `main` aus? Warum benötigt sie, obwohl es keinen Verweis auf eine Datei auf der Festplatte gibt, eine eigene Klausel `throws IOException`? Nun, eine Ausnahme ist wie eine heiße Kartoffel. Wenn Sie eine haben, müssen Sie sie entweder verspeisen (wie das geht, zeigt [Kapitel 12](#)) oder Sie verwenden eine `throws`-Klausel, um die Ausnahme irgendwohin weiterzureichen. Wenn Sie eine Ausnahme mit `throw` weiterreichen, lösen Sie das

Problem nicht, sondern erreichen damit nur, dass sich jemand an anderer Stelle damit beschäftigen muss.

Der Konstruktor in [Listing 10.2](#) wirft eine `IOException`, aber wohin wird sie nun geworfen (weitergereicht)? Wer im Code übernimmt die Verantwortung für die problematische `IOException`? Wer hat den Konstruktor in [Listing 10.2](#) aufgerufen? Das war die Methode `main` in [Listing 10.3](#) – hier geht es um den Aufruf des Konstruktors `TeamFrame`. Da der Konstruktor `TeamFrame` in [Listing 10.3](#) seine heiße Kartoffel der Methode `main` zuwirft, muss sich diese Methode darum kümmern. [Listing 10.3](#) zeigt, dass sie dies dadurch »erledigt«, dass sie die `IOException` wirft (indem sie über eine eigene Klausel `throws IOException` verfügt). So funktionieren in Java-Programmen `throws`-Klauseln.



Wenn eine Methode eine andere Methode aufruft, die eine `throws`-Klausel enthält, muss es in der aufrufenden Methode Code geben, der mit der Ausnahme umgeht. Was Sie mit Ausnahmen machen können, steht in [Kapitel 12](#).



Möglicherweise fallen dem scharfsinnigen *Für-Dummies*-Leser genau jetzt eine oder zwei Fragen ein. »Wenn eine Methode `main` eine `throws`-Klausel enthält, muss sich jemand anders um die Ausnahme in der `throws`-Klausel kümmern. Wer aber ruft die Methode `main` auf? Wer kümmert sich um die `IOException` in der `throws`-Klausel von [Listing 10.3](#)? Die Antwort lautet, dass die Java Virtual Machine (oder JVM, das Ding, das Ihren gesamten Java-Code ausführt) die Methode `main` aufgerufen hat. Es ist also die JVM, die sich in [Listing 10.3](#) um die `IOException` kümmert. Wenn das Programm Schwierigkeiten hat, die Datei `Hankees.txt` zu lesen, wird die letzte Verantwortung an die JVM übertragen. Die JVM nimmt sich dann der Sache an, indem sie eine Fehlermeldung ausgibt und die Ausführung des Programms beendet.

Etwas statisch machen (den Teamdurchschnitt herausfinden)

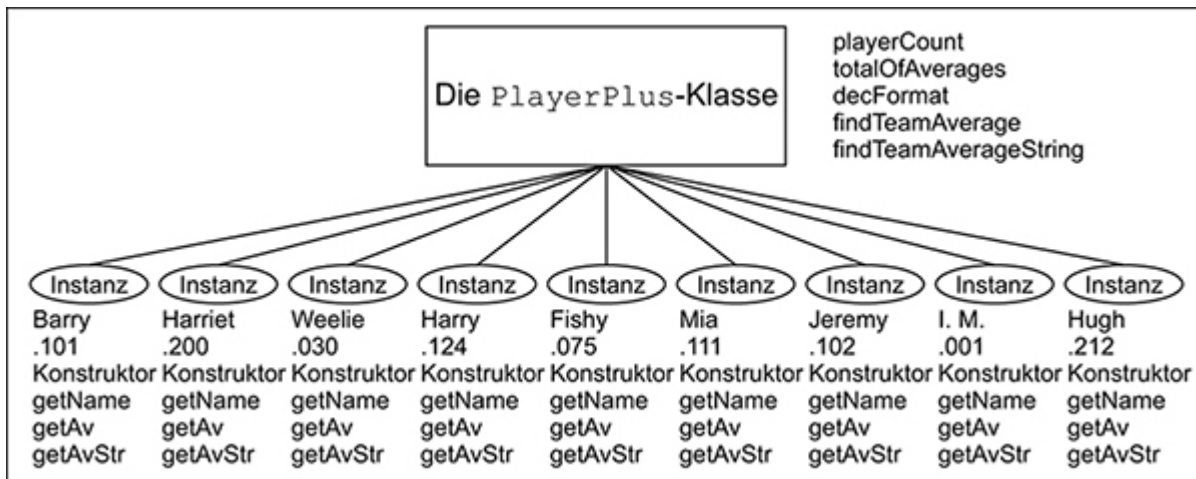
Bei Ihrer Beschäftigung mit dem Code in den [Listings 10.1](#) bis [10.3](#) kommen Sie auf die Idee, dass es doch ganz interessant sein könnte, den Schlagdurchschnitt des gesamten Teams herauszufinden. Die Hankees in [Abbildung 10.1](#) haben nur einen Durchschnitt von ungefähr $,106$ – sie sollten deshalb viel intensiver trainieren. Während sich die Spieler draußen auf dem Spielfeld tummeln, müssen Sie ein technisches Problem lösen.

In den [Listings 10.1](#) bis [10.3](#) gibt es drei Klassen: die Klasse `Player` und zwei weitere Klassen, die helfen, Daten der Klasse `Player` anzuzeigen. Wo also sind in dieser undurchsichtigen Anhäufung von Klassen die Variablen abgeblieben, die den Teamdurchschnitt wie eine Strichliste speichern?

- ✓ Es macht keinen Sinn, zählende Variablen in einer der Klassen unterzubringen, die für die Anzeige zuständig sind (die Klassen `TeamFrame` und `ShowTeamFrame`). Schließlich hat dieser Durchschnittswert mit Spielern, Teams und Baseball zu tun. Die anzeigenden Klassen kümmern sich um Fenster und nicht darum, Baseball zu spielen.
- ✓ Sie fühlen sich nicht wohl bei dem Gedanken, den Teamdurchschnitt in einer Instanz der Klasse `Player` abzulegen, weil eine solche Instanz immer nur einen Spieler des Teams darstellt. Was haben einzelne Spieler damit zu tun, allgemeine Daten des Teams zu speichern? Klar, Sie können dafür sorgen, dass der Code funktioniert, aber das wäre keine elegante Lösung des Problems.

Und dann entdecken Sie das Schlüsselwort `static` (deutsch *statisch*). Alles, was als statisch deklariert wird, gehört einer ganzen Klasse und nicht einer bestimmten Instanz der Klasse. Wenn Sie das Feld `totalOfAverages` (deutsch

Gesamtdurchschnitt) statisch deklarieren, erstellen Sie nur eine Kopie dieses Feldes. Diese Kopie steht der gesamten Klasse `Player` zur Verfügung. Unabhängig davon, wie viele Instanzen der Klasse `Player` existieren – eine, neun oder keine –, es gibt immer nur ein Feld `totalOfAverages`. Und da Sie schon einmal dabei sind, legen Sie gleich noch weitere statische Felder (`playerCount` und `decFormat`) und statische Methoden (`findTeamAverage` und `findTeamAverageString`) an. Wenn Sie wissen wollen, was ich damit meine, schauen Sie sich [Abbildung 10.5](#) an.



[Abbildung 10.5](#): Statische und nicht statische Felder und Methoden

Es kommt sicherlich Ihrer Vorliebe für Unterklassen entgegen, wenn Sie den Code für die teamweiten Daten in einer Unterklasse der Klasse `Player` bereitstellen. [Listing 10.4](#) enthält den entsprechenden Code.

```
import java.text.DecimalFormat;
public class PlayerPlus extends Player {
    private static int playerCount = 0;
    private static double totalOfAverages = .000;
    private static DecimalFormat decFormat =
        new DecimalFormat(
);
    static {
        decFormat.setMaximumIntegerDigits(0);
```

```

        decFormat.setMaximumFractionDigits(3);
        decFormat.setMinimumFractionDigits(3);
    }
    public PlayerPlus(String name, double average) {
        super(name, average);
        playerCount++;
        totalOfAverages += average;
    }
    public static double findTeamAverage() {
        return totalOfAverages / playerCount;
    }

    public static String findTeamAverageString() {
        return decFormat.format
            (totalOfAverages / playerCount)
;
    }
}

```

[Listing 10.4](#): Einen Schlagdurchschnitt des Teams erstellen

Warum gibt es da so viel Statisches?

Vielleicht ist es Ihnen aufgefallen, dass im Code von [Listing 10.4](#) das Wort `static` sehr häufig vorkommt. Das liegt daran, dass so gut wie alles in diesem Code zur gesamten Klasse `Player Plus` und nicht zu einzelnen Instanzen dieser Klasse gehört. Das muss auch so sein, denn Methoden wie `playerCount` (die Anzahl der Spieler, die das Team bilden) gehören nicht zu einzelnen Spielern, und es wäre mehr als unsinnig, wenn jedes `PlayerPlus`-Objekt seine eigene Anzahl verwalten müsste. (»Ich weiß, wie viele Spieler ich bin. Ich bin immer nur ein Spieler.«) Wenn Sie neun einzelne `playerCount`-Felder hätten, würden Sie dann in jedem Feld die Zahl 1 speichern (was sinnlos ist) oder gäbe es neun verschiedene Kopien der Anzahl (was verschwenderisch und fehleranfällig wäre)? Aber dadurch, dass Sie `playerCount` statisch machen, sorgen Sie dafür, dass es `playerCount` nur an einer einzigen Stelle gibt.

Dieselben Gründe gelten für `totalOfAverages`. Letztendlich speichert das Feld `totalOfAverages` die Summe der *Batting Averages* der Spieler. Sie beträgt bei den Hankees , 956. Erst wenn eine der Methoden `findTeamAverage` oder `findTeamAverageString` aufgerufen wird, berechnet der Computer den *Batting Average* des gesamten Hankees-Teams.

Auch die Methoden `findTeamAverage` und `findTeamAverageString` müssen statisch sein. Ohne das Wort `static` gäbe es neun `findTeamAverage`-Methoden – eine für jede Instanz der Klasse `PlayerPlus`. Auch das wäre ziemlich unsinnig. Jede Instanz enthielte den Code, um `totalOfAverages/playerCount` zu berechnen, und alle neun Berechnungen würden dasselbe Ergebnis liefern.



Im Allgemeinen sollte jede Aufgabe, die allen Instanzen gemeinsam ist (und die für jede Instanz dasselbe Ergebnis liefert), als statische Methode kodiert werden.



Konstruktoren sind niemals statisch.

Das statische Initialisierungsprogramm

In [Listing 10.4](#) ist das Feld `decFormat` statisch. Das macht Sinn, weil `decFormat` dafür sorgt, dass `totalOfAverages/playerCount` schön anzuschauen ist, und die beiden Felder, die den Ausdruck bilden, sind ebenfalls statisch. Letztendlich benötigt der Code nur ein Element, um Zahlen zu formatieren. Wenn mehrere Zahlen darauf warten, optisch ansprechend dargestellt zu werden, ist ein `decFormat`-Element, das zur ganzen Klasse gehört, in der Lage, alle Zahlen zu formatieren. Das Erstellen von `decFormat` für jeden einzelnen Spieler ist nicht nur wenig elegant, sondern auch vergeudete Zeit.

Aber dadurch, dass Sie `decFormat` statisch deklarieren, stehen Sie plötzlich vor einem kleinen Problem. Sie müssen, um die

Formatierung zu aktivieren, Methoden wie `decFormat.setMaximumIntegerDigits(0)` aufrufen. Sie dürfen diese Methode nicht irgendwo in der Klasse `PlayerPlus` unterbringen. So ist zum Beispiel der folgende Code schlecht, ungültig, illegal und auf keinen Fall für Java zu gebrauchen:

```
// DIES IST MISERABLER CODE:
public class PlayerPlus extends Player {
    private static DecimalFormat decFormat =
        new DecimalFormat(
    );
    decFormat.setMaximumIntegerDigits(0);    // Schlecht!
    decFormat.setMaximumFractionDigits(3);    // Schlecht!
    decFormat.setMinimumFractionDigits(3);    // Schlecht!
```

Schauen Sie sich die Beispiele der vorherigen Kapitel an. Dort lasse ich Methodenaufrufe niemals so in der Luft hängen, wie es bei obigem miserablen Codebeispiel der Fall ist. Ich rufe in [Listing 10.1](#) die Methode `setMaximumIntegerDigits` aus dem Körper der Methode `getAverageString` heraus auf. Und das geschieht nicht zufällig. Die Java-Regeln legen strenge Maßstäbe daran an, wo Methodenaufrufe im Code platziert werden müssen, und es ist überhaupt nicht zulässig, einen Methodenaufruf direkt in der Definition einer Klasse unterzubringen.

Wo können Sie also in [Listing 10.4](#) die Aufrufe von `setMax` und `setMin` hinpacken? Sie können sie so in den Körper der Methode `findTeamAverageString` schreiben, wie ich diese Aufrufe in [Listing 10.1](#) im Körper der Methode `getAverageString` untergebracht habe. Nun könnte es aber so sein, dass ich dadurch, dass sich die Methodenaufrufe im Körper der Methode `findTeamAverageString` befinden, `decFormat` daran hindere, statisch zu sein. Schließlich könnte ein Programmierer `findTeamAverageString` mehrmals aufrufen, was dann auch für `decFormat.setMaximumIntegerDigits(0)` gilt. Das wäre eine sinnlose Aktion. Die gesamte Klasse `PlayerPlus` enthält nur ein `decFormat`-Feld, dessen `MaximumIntegerDigits`-

Wert immer 0 ist. Also vermeiden Sie es,
`MaximumIntegerDigits(0)` ständig neu festzulegen.

Die beste Lösung sieht so aus, dass Sie in diesem Abschnitt die Zeilen des miesen Codes nehmen und in einem *statischen Initialisierer* unterbringen. Dann werden schlechte Codezeilen in gutem Code zu guten Zeilen (siehe [Listing 10.4](#)). Bei einem statischen Initialisierer handelt es sich um einen Block, dem das Wort `static` vorangestellt ist. Java führt die Befehle eines statischen Initialisierers einmal für die gesamte Klasse aus. Und das ist genau das, was Sie sich für etwas wünschen, das »statisch« genannt wird.

Den Teamdurchschnitt anzeigen

Vielleicht fällt Ihnen ein Muster auf, denn wenn Sie Code für eine Klasse erstellen, besteht dieser im Allgemeinen aus zwei Teilen. Der erste definiert die Klasse und der zweite verwendet die Klasse. (Zum Verwenden einer Klasse gehört, dass Sie deren Konstruktor aufrufen, auf die nicht privaten Felder der Klasse verweisen, die Methoden der Klasse aufrufen und so weiter.) [Listing 10.4](#), das weiter vorn in diesem Kapitel steht, enthält Code, der die Klasse `PlayerPlus` definiert, während [Listing 10.5](#) Code enthält, der diese `PlayerPlus`-Klasse verwendet.

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
import javax.swing.JFrame;
import javax.swing.JLabel;
import java.awt.GridLayout;
@SuppressWarnings("serial")
public class TeamFrame extends JFrame {
    public TeamFrame() throws IOException {
        PlayerPlus player;
        Scanner hankeesData =
            new Scanner(new File("Hankees.txt
"));
        for (int num = 1; num <= 9; num++) {
            player =
                new PlayerPlus(hankeesData.nextLine(),
```

```

        hankeesData.nextDouble (
    ));

    hankeesData.nextLine ();

    addPlayerInfo (player);
}

add(new JLabel());
add(new JLabel("  -----"));
add(new JLabel(
    "Der Schlagdurchschnitt des Teams:"));
add(new JLabel(
    PlayerPlus.findTeamAverageString()));
setTitle("Die Hankees");
setLayout(new GridLayout(11, 2, 20, 3));
setDefaultCloseOperation(EXIT_ON_CLOSE);
pack();
setVisible(true);
hankeesData.close();
}

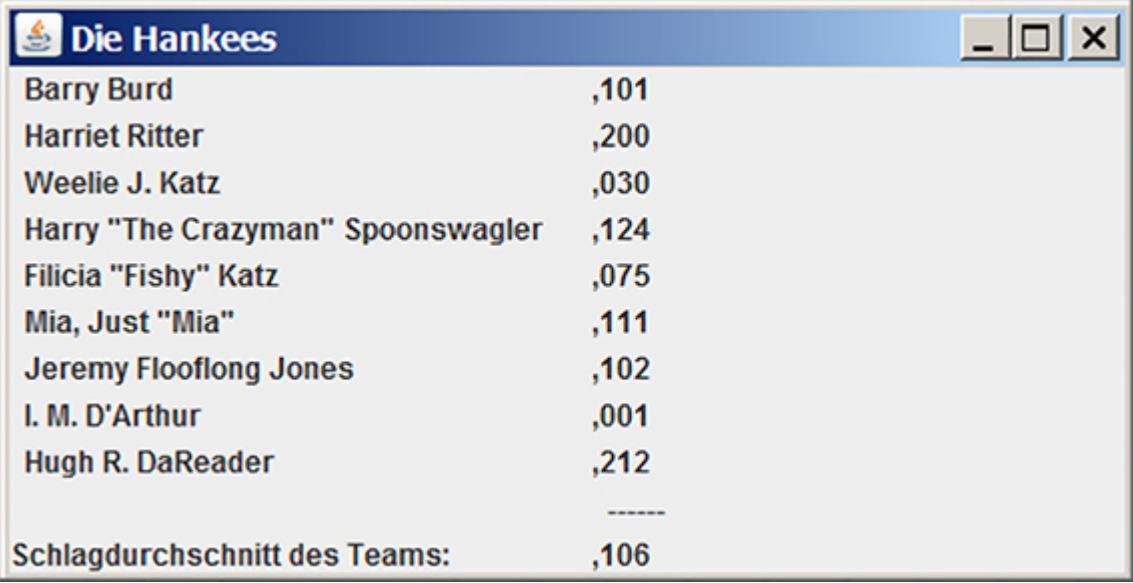
void addPlayerInfo(PlayerPlus player) {
    add(new JLabel("  " + player.getName()));
    add(new JLabel(player.getAverageString()));
}
}

```

[Listing 10.5](#): Den Code aus [Listing 10.4](#) verwenden

Wenn Sie den Code in [Listing 10.5](#) ausführen wollen, benötigen Sie eine Klasse mit einer Methode `main`. Diese Aufgabe übernimmt die Klasse `ShowTeamFrame` aus [Listing 10.3](#).

[Abbildung 10.6](#) zeigt, was der Code in [Listing 10.5](#) ausgibt, wenn Sie ihn ausführen. Damit Sie das Programm erfolgreich ablaufen lassen können, muss die Datei `Hankees.txt` aus [Abbildung 10.2](#) vorhanden sein. Der Code in [Listing 10.5](#) ist eine fast vollständige Kopie des Codes in [Listing 10.2](#). Das einzig Neue sind in [Listing 10.5](#) die fett gedruckten Bestandteile des Codes.



Barry Burd	,101
Harriet Ritter	,200
Weelie J. Katz	,030
Harry "The Crazyman" Spoonswagler	,124
Filicia "Fishy" Katz	,075
Mia, Just "Mia"	,111
Jeremy Flooflong Jones	,102
I. M. D'Arthur	,001
Hugh R. DaReader	,212

Schlagdurchschnitt des Teams:	,106

Abbildung 10.6: Der Code aus [Listing 10.5](#) wird ausgeführt.

In [Listing 10.5](#) ist das `GridLayout` um zwei zusätzliche Zeilen erweitert worden: eine Zeile, die als Trennzeile dient, und eine zweite Zeile für den Teamdurchschnitt der Hankees. In beiden Zeilen gibt es jeweils zwei neue `Label`-Objekte.

✓ **Die leere Zeile enthält ein leeres Label und ein Label mit einer gestrichelten Linie.** Das leere Label dient als Platzhalter. Wenn Sie einem `GridLayout` Komponenten hinzufügen, geschieht dies zeilenweise. Dabei wird am linken Ende einer Zeile angefangen und die Zeile nach rechts abgearbeitet. Ohne das leere Label würde die gestrichelte Linie am linken Ende der Zeile unter den Namen der Spieler erscheinen.

✓ **Die andere Zeile enthält ein Label, das die Wörter Schlagdurchschnitt des Teams anzeigt, und ein zweites Label für die Zahl ,106.** Der Methodenaufruf, der die Zahl ,106 erhält, ist interessant. Der Aufruf sieht so aus:

```
PlayerPlus.findTeamAverageString()
```

Dieser Methodenaufruf hat die Form

KlassenName.methodenName

Das ist neu! Ich weise in früheren Kapiteln darauf hin, dass Sie einen Methodenaufruf normalerweise mit dem Namen eines

Objekts und nicht mit dem einer Klasse beginnen. Warum nun benutze ich hier einen Klassennamen? Wenn Sie eine statische Methode aufrufen, setzen Sie vor den Namen der Methode den Namen der Klasse, die die Methode enthält. Dasselbe gilt, wenn Sie ein statisches Feld einer anderen Klasse referenzieren. Der Grund: Die ganze Klasse, die ein statisches Feld oder eine statische Methode deklariert, ist Eigentümer dieses Feldes beziehungsweise der Methode. Um also auf eine statische Methode oder ein statisches Feld zu verweisen, beginnen Sie den Namen des Feldes oder der Methode mit dem Namen der Klasse.



Wenn Sie auf ein statisches Feld oder eine statische Methode verweisen, können Sie auch schummeln und anstelle des Klassennamens den Namen eines Objekts verwenden. Wenn Sie zum Beispiel in [Listing 10.5](#) einige Befehle geschickt neu anordnen, können Sie den Ausdruck `player.findTeamAverageString()` verwenden.

»static« ist ein alter Hut

Dieser Abschnitt macht ziemlich viel Wind um statische Felder und Methoden, aber statische Elemente tauchen schon ganz früh in diesem Buch auf. So stellt zum Beispiel [Kapitel 3](#) `System.out.println` vor. Der Name *System* verweist auf eine Klasse und bei *out* handelt es sich um ein statisches Feld dieser Klasse. Aus diesem Grund verwende ich in [Kapitel 4](#) und danach das Schlüsselwort `static`, um das Feld `out` zu importieren:

```
import static java.lang.System.out;
```

In Java kann von überall her auf statische Felder und Methoden zugegriffen werden. Wenn so etwas im Code eines anderen Programmierers deklariert wird und Sie das so Deklarierte in Ihrem Code verwenden wollen, müssen Sie sich darüber keine Gedanken machen. Aber wenn Sie Ihre Felder und Methoden deklarieren und vor der Frage stehen, ob etwas statisch sein soll, müssen Sie schon ein wenig intensiver über die richtige Entscheidung nachdenken.



Ich habe das Wort `static` in diesem Buch zum ersten Mal ernsthaft in [Listing 3.1](#) verwendet. Dieses Schlüsselwort taucht in jeder `main`-Methode auf (von denen es in den Listings dieses Buches ebenfalls viele gibt). Warum nun muss `main` statisch sein? Erinnern Sie sich daran, dass nicht statische Komponenten zu Objekten und nicht zu Klassen gehören? Wenn die Methode `main` nicht statisch ist, gibt es sie erst, wenn ein Objekt existiert. Wenn Sie aber ein Java-Projekt anfangen, haben Sie noch keine Objekte erstellt. Die Befehle, die in der Methode `main` ausgeführt werden, erstellen die Objekte. Damit stehen Sie ohne eine *statische* Methode `main` vor dem Huhn-Ei-Problem.

Achtung Statisches – mit Vorsicht genießen!

Als ich anfang, Java-Code zu schreiben, hatte ich einen immer wiederkehrenden Traum von einer Fehlermeldung. Die Meldung lautete `non static variable or method cannot be referenced from a static context` (*nicht statische Variable oder Methode darf aus einem statischen Kontext heraus nicht referenziert werden*). Und jedes Mal, wenn ich diese Fehlermeldung zu lesen bekomme, war ich aufs Neue so perplex, dass sich diese Meldung in mein Unterbewusstsein eingebrannt hat.

Heute weiß ich, warum ich diese Fehlermeldung so oft erhalten habe. Ich kann sie bei Bedarf sogar künstlich auslösen. Aber mir läuft immer noch ein Schauer den Rücken runter, wenn ich sie auf meinem Bildschirm sehe.

Bevor Sie verstehen können, warum es zu dieser Meldung kommt und wie sich das Problem lösen lässt, müssen Sie ein paar Fachbegriffe kennenlernen. Wenn ein Feld oder eine Methode nicht statisch ist, wird es beziehungsweise sie auch als *nicht statisch* oder *non static* bezeichnet. (Eine echte Überraschung,

oder?) Dies sollte nun dabei helfen, mindestens zwei Wege zu beschreiben, um die gefürchtete Fehlermeldung hervorzurufen:

✓ Bringen Sie irgendwo im Code `Klasse.nonstaticDing` unter.

✓ Bringen Sie irgendwo in einer statischen Methode `nonstaticDing` unter.

In beiden Fällen bringen Sie sich in Schwierigkeiten. Sie nehmen etwas, das zu einem Objekt (dem nicht statischen Ding) gehört, und bringen es an einer Stelle unter, an dem keine Objekte in Sichtweite sind.

Nehmen wir zum Beispiel die erste der beiden Situationen, die ich gerade beschrieben habe. Um zu sehen, wie es zu diesem Unglück kommt, schauen Sie sich noch einmal [Listing 10.5](#) an.

Ändern Sie am Ende des Listings `player.getName()` in `Player.getName()`. Das ist alles. Was könnte `Player.getName()` bedeuten? Wenn überhaupt, bedeutet es: »Rufe die Methode `getName` auf, die zur gesamten Klasse `Player` gehört.« Werfen Sie noch einmal einen Blick auf [Listing 10.1](#). Die Methode `getName` ist nicht statisch. Jede Instanz der Klasse `Player` (oder `PlayerPlus`) enthält eine Methode `getName`. Keine dieser Methoden gehört der gesamten Klasse, weshalb der Aufruf `Player.getName` unsinnig ist.

Um in die zweite Situation hineinzuschnuppern, sehen Sie sich noch einmal [Listing 10.4](#) an. Entfernen Sie (zu Beginn des Listings) in der Deklaration des Feldes `decFormat` das Wort `static`. Dies macht aus `decFormat` ein nicht statisches Feld. Und plötzlich erhält jeder Spieler des Teams ein eigenes `decFormat`-Feld.

Nun, alles geht so lange gut, bis der Computer die Methode `findTeamAverageString` erreicht. Diese statische Methode enthält vier `decFormat.DiesUndDas`-Befehle. Und erneut sind Sie gezwungen zu fragen, was ein Befehl dieser Art bedeutet. Die Methode `findTeamAverageString` gehört zu keiner besonderen Instanz. (Die Methode ist statisch, weshalb die

gesamte Klasse `PlayerPlus` nur eine einzige Methode `findTeamAverageString` kennt.) Aber so, wie Sie den Code gerade verunstaltet haben, gibt es für das gute alte `decFormat` ohne eine Referenz auf ein bestimmtes Objekt keine Bedeutung mehr. Sie verweisen also wieder aus dem Kontext einer statischen Methode heraus auf das nicht statische Feld `decFormat`.

Mit Variablen experimentieren

Während meiner Zeit als Student saß ich an einem schönen Sommertag in der Sonne, trödelte herum und sprach mit einer Kommilitonin. Ich glaube, ihr Name war Janine. »Wo kommst du her?«, fragte ich sie. »Mars«, antwortete sie. Sie machte eine Pause und wartete, ob ich nachhaken würde.

Wie es sich herausstellte, kam Janine aus Mars in Pennsylvania, einer kleinen Stadt, die ungefähr 20 Meilen nördlich von Pittsburgh liegt. Was will ich damit sagen? Es geht darum, dass die Bedeutung eines Namens vom Kontext abhängt. Wenn Sie sich nördlich von Pittsburgh befinden und nach dem Weg fragen und als Ziel *Mars* angeben, erhalten Sie vielleicht eine brauchbare Antwort. Wenn Sie aber dieselbe Frage in Manhattan stellen, werden Sie wahrscheinlich schief angesehen. (Okay, da ich Manhattan kenne, weiß ich, dass die Leute dort Sie höchstwahrscheinlich ignorieren werden.)

Natürlich wissen die Leute, die in Mars, Pennsylvania, leben, genau, dass der Name ihrer Stadt zu Missverständnissen führen kann, aber sie haben sich daran gewöhnt. Aber gerade an diesem Beispiel zeigt sich, wie sehr die Bedeutung eines Namens davon abhängt, wo Sie ihn verwenden. Sicherlich denken die meisten Menschen an den Roten Planeten, wenn sie den Namen *Mars* hören, aber in Pennsylvania leben ein paar Leute, denen bei diesem Namen nur die Einkaufsmöglichkeiten einfallen, die ihnen der Ort *Mars* bietet. Für die Menschen in Pennsylvania hat der Name *Mars* zwei Bedeutungen. In Java würde dies wohl so aussehen: `Mars` und `Planet.Mars`.

Eine Variable richtig platzieren

Die [Listings 10.6](#) und [10.7](#) enthalten Ihr erstes Experiment. Im Code werden die Unterschiede zwischen Variablen, die innerhalb von Methoden deklariert werden, und Variablen, die außerhalb von Methoden deklariert werden, optisch hervorgehoben.

```
import static java.lang.System.out;
class EnglishSpeakingWorld {
    String mars = "    Roter Planet";
    void visitPennsylvania() {
        out.println("»visitPA« wird ausgeführt:");
        String mars = "    Janines Heimatstadt";
        out.println(mars);
        out.println(this.mars);
    }
}
```

[Listing 10.6](#): Zwei Bedeutungen von »Mars«

```
import static java.lang.System.out;
public class GetGoing {

    public static void main(String args[]) {

        out.println("»main« wird ausgeführt:");

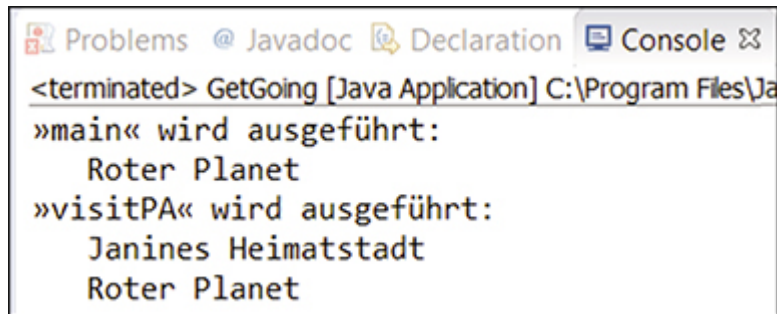
        EnglishSpeakingWorld e =
            new EnglishSpeakingWorld();

        //out.println(mars);    cannot resolve symbol
        out.println(e.mars);
        e.visitPennsylvania();
    }
}
```

[Listing 10.7](#): Den Code in [Listing 10.6](#) aufrufen

[Abbildung 10.7](#) zeigt, was der Code der [Listings 10.6](#) und [10.7](#) ausgibt, wenn er ausgeführt wird. [Abbildung 10.8](#) gibt die Struktur des Codes grafisch wieder. Die Methode `main` erstellt in der Klasse `GetGoing` eine Instanz der Klasse `EnglishSpeakingWorld`. Die Variable `e` verweist auf die neue Instanz. Bei der neuen Instanz handelt es sich um ein Objekt mit

einer Variablen, die im Objekt `mars` genannt wird. Diese Variable `mars` hat den Wert "Roter Planet". Die `mars`-Variable "Roter Planet" ist ein Feld.



```
<terminated> GetGoing [Java Application] C:\Program Files\Ja
»main« wird ausgeführt:
    Roter Planet
»visitPA« wird ausgeführt:
    Janines Heimatstadt
    Roter Planet
```

Abbildung 10.7: Der Code in den Listings 10.6 und 10.7 wird ausgeführt.

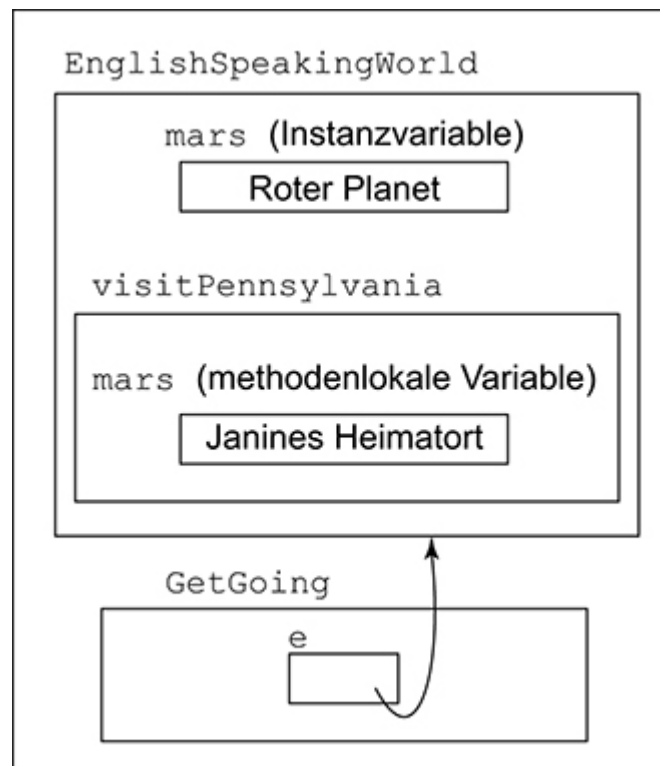


Abbildung 10.8: Die Struktur der Listings 10.6 und 10.7



Wenn Sie das Feld `mars` anders beschreiben wollen, bezeichnen Sie es als *Instanzvariable*, weil diese Variable

`mars` (deren Wert "Roter Planet" ist) zu einer Instanz der Klasse `EnglishSpeakingWorld` gehört. Im Gegensatz hierzu fallen statische Felder (wie die Felder `playerCount`, `totalOfAverages` und `decFormat` in [Listing 10.4](#)) in die Kategorie *Klassenvariablen*. So ist zum Beispiel `playerCount` in [Listing 10.4](#) eine Klassenvariable, weil eine Kopie dieser Variablen der gesamten Klasse `PlayerPlus` gehört.

Werfen Sie nun in [Listing 10.7](#) einen Blick auf die Methode `main`. Sie dürfen in dieser Methode der Klasse `GetGoing` nicht `out.println(mars)` schreiben, denn es ist auf keinen Fall zulässig, ohne zusätzliche Informationen auf die Variable `mars` zu verweisen. Die Variable `mars`, um die es mir im letzten Absatz geht, gehört zum Objekt `EnglishSpeakingWorld` und nicht zur Klasse `GetGoing`.

Natürlich können Sie in der Methode `main` der Klasse `GetGoing` `e.mars` schreiben, weil die Variable `e` auf Ihr Objekt `EnglishSpeakingWorld` verweist.

Im unteren Bereich des Codes wird die Methode `visitPennsylvania` aufgerufen. Wenn Sie sich in `visitPennsylvania` befinden, stehen Sie vor einer anderen Deklaration der Variablen `mars`, deren Wert "Janines Heimatstadt" lautet. Diese `mars`-Variable wird als *methodenlokale Variable* bezeichnet, weil sie nur zu einer einzigen Methode gehört – der Methode `visitPennsylvania`.

Sie haben nun zwei Variablen, die beide den Namen `mars` tragen. Eine davon, die ein Feld ist, hat den Wert "Roter Planet". Die andere `mars`-Variable, eine methodenlokale Variable, hat den Wert "Janines Heimatstadt". Wenn Sie im Code das Wort `mars` verwenden, provozieren Sie die Frage, welche der beiden Variablen Sie meinen.

Wenn Sie Pennsylvania besuchen, gewinnt "Janines Heimatstadt". Und wenn Sie sich dort befinden, müssen Sie wie ein Einheimischer denken. Wenn Sie Code ausführen, der in

der Methode `visitPennsylvania` steht, lösen Sie Namenskonflikte dadurch, dass Sie über die methodenlokalen Variablen gehen – Variablen, die direkt in der Methode `visitPennsylvania` deklariert werden.

Wie sieht es also aus, wenn Sie sich in Pennsylvania befinden und auf den Himmelskörper mit den zwei Monden verweisen müssen? Oder um genau zu sein: Wie kann Code, der sich in der Methode `visitPennsylvania` befindet, auf das Feld mit dem Wert "Roter Planet" verweisen? Ganz einfach, verwenden Sie `this.mars`. Das Wort `this` verweist auf das Objekt, das den gesamten Code enthält (und nicht auf eine Methode in diesem Code). Dieses Objekt, das eine Instanz der Klasse `EnglishSpeakingWorld` ist, enthält ein dickes, fettes Feld `mars`, und der Wert dieses Feldes ist "Roter Planet". Das ist der Weg, wie Sie Code dazu zwingen können, sich auch außerhalb der Methode umzublicken, in der er steht – indem Sie das Java-Schlüsselwort `this` verwenden. (In [Kapitel 9](#) stehen weitere Informationen zu `this`.)

Einer Variablen aufzeigen, wohin sie gehört

Als ich vor einigen Jahren in Milwaukee, Wisconsin, lebte, habe ich regen Gebrauch von den Geldautomaten der lokalen Bank gemacht. Zu diesem Zeitpunkt begann man gerade damit, Automaten dieser Art zu standardisieren. Das lokale System dieser Automaten hieß *ZAIT*, was die Abkürzung von *Zentraler Abruf Ihres Tagesgeldbedarfs* war.

Ich erinnere mich an eine Reise mit dem Auto durch Kalifornien. Irgendwann wurde ich hungrig und stoppte, um mir etwas zum Essen zu kaufen. Da ich kein Bargeld mehr besaß, fragte ich den Mitarbeiter einer Tankstelle: »Können Sie mir sagen, wo hier in der Gegend die nächste ZAIT-Maschine steht?«

Sie sehen, dass ein Name, der an einem Ort problemlos verstanden wird, anderswo zu ziemlichen Missverständnissen

führen kann. Mit den [Listings 10.8](#) und [10.9](#) möchte ich diesen Punkt (mit mehr als nur einer Anekdote über Geldautomaten) verdeutlichen.

```
import static java.lang.System.out;
class EnglishSpeakingWorld2 {
    String mars;
    void visitIdaho() {
        out.println("»visitID« wird ausgeführt:");
        mars = "    Roter Planet";
        String atomicCity = "    Einwohner: 25";
        out.println(mars);
        out.println(atomicCity);
    }
    void visitNewJersey() {
        out.println("»visitNJ« wird ausgeführt:");
        out.println(mars);
        //out.println(atomicCity);
        //    cannot resolve symbol
    }
}
```

[Listing 10.8](#): Eine Geschichte aus Atomic City

```
public class GetGoing2 {
    public static void main(String args[]) {
        EnglishSpeakingWorld2 e =
                                new EnglishSpeakingWorld2();
        e.visitIdaho();
        e.visitNewJersey();
    }
}
```

[Listing 10.9](#): Den Code in [Listing 10.8](#) aufrufen

[Abbildung 10.9](#) zeigt, was ausgegeben wird, wenn Sie den Code in den [Listings 10.8](#) und [10.9](#) ausführen. [Abbildung 10.10](#) stellt die Struktur dieses Codes dar. Der Code von `EnglishSpeakingWorld2` enthält zwei Variablen. Die eine, die Variable `mars`, wird nicht innerhalb einer Methode deklariert. Sie ist ein Feld. Bei der anderen Variablen, `atomicCity`, handelt es sich um eine methodenlokale Variable, die in der Methode `visitIdaho` deklariert wird.



```
<terminated> GetGoing2 [Java Application] C:\Program Files\J  
»visitID« wird ausgeführt:  
    Roter Planet  
    Einwohner: 25  
»visitNJ« wird ausgeführt:  
    Roter Planet
```

Abbildung 10.9: Der Code der Listings 10.8 und 10.9 wird ausgeführt.

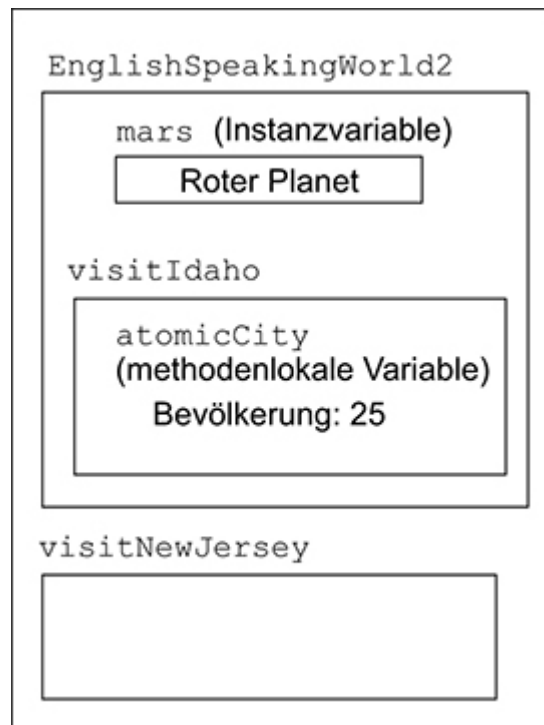


Abbildung 10.10: Die Struktur des Codes der Listings 10.8 und 10.9

Achten Sie darauf, wo in Listing 10.8 Variablen verwendet werden können oder auch nicht. Wenn Sie versuchen, die Variable `atomicCity` in der Methode `visitNewJersey` zu verwenden, erhalten Sie eine Fehlermeldung. Diese Meldung besagt wörtlich `cannot resolve symbol` (was bedeutet, dass ein Symbol nicht aufgelöst werden kann). Im übertragenen Sinne lautet die Botschaft: »Hallo, mein Freund, Atomic City liegt in Idaho und

nicht in New Jersey.« Technisch gesehen besagt die Meldung, dass die methodenlokale Variable `atomicCity` nur der Methode `visitIdaho` zur Verfügung steht, weil die Variable dort deklariert worden ist. Aber in dieser Methode können Sie die Variable `atomicCity` uneingeschränkt verwenden.

Und was ist mit dem Mars? Haben Sie Ihren alten Freund, diesen eiskalten Planeten, vergessen? Nun, die beiden Methoden `visitIdaho` und `visitNewJersey` sind in der Lage, auf die Variable `mars` zuzugreifen. Dies geht, weil es sich bei dieser Variablen um ein Feld handelt. Sie wird zwar im Code der Klasse `EnglishSpeakingWorld2` deklariert, aber nicht innerhalb einer bestimmten Methode. (Wir können wohl davon ausgehen, dass die Bewohner Idahos und New Jerseys schon vom Planeten Mars gehört haben.)

Der Lebenszyklus des Feldes `mars` besteht aus drei separaten Phasen:

- ✓ Wenn die Klasse `EnglishSpeakingWorld2` zum ersten Mal zum Leben erweckt wird, sieht der Computer `String mars` und reserviert Platz für das Feld `mars`.
- ✓ Wenn die Methode `visitIdaho` ausgeführt wird, weist sie dem Feld `mars` den Wert "Roter Planet" zu. (Die Methode `visitIdaho` sorgt auch dafür, dass der Wert ausgegeben wird, den das Feld `mars` hat.)
- ✓ Wenn die Methode `visitNewJersey` ausgeführt wird, gibt sie noch einmal den Wert aus, den das Feld `mars` hat.

Auf diese Weise wird der Wert des Feldes `mars` von einer Methode zur nächsten weitergegeben.

Parameter übergeben

Es gibt für eine Methode verschiedene Möglichkeiten, um mit einem anderen Teil Ihres Programms zu kommunizieren. Einer davon führt über die Parameterliste der Methode. Indem Sie eine Parameterliste verwenden, können Sie bei einem Aufruf der

Methode auf die Schnelle Informationen an eine andere Methode weitergeben.

Stellen Sie sich vor, dass die Informationen, die Sie an die andere Methode übergeben wollen, in einer der Variablen des Programms gespeichert werden. Was wird dann die Methode wohl mit dieser Variablen machen? Die folgenden Abschnitte bieten ein paar interessante Fallstudien an.

Übergabe als Wert

Meine Suche im Web hat ergeben, dass der Ort Smackover in Arkansas 2.232 Einwohner hat. Doch meine Zahlen sind nicht aktuell. Erst gestern konnte Dora Kermongoos im Smackover General Hospital ein freudiges Ereignis feiern – die Geburt ihrer gesunden, blauäugigen Tochter. Damit wuchs die Einwohnerzahl auf 2.233.

[Listing 10.10](#) enthält ein sehr schlechtes Programm. Es sollte eine Variable, in der die Einwohnerzahl Smackovers gespeichert ist, um 1 erhöhen, aber das Programm funktioniert nicht. Schauen Sie sich [Listing 10.10](#) einmal an.

```
public class TrackPopulation {
    public static void main(String args[]) {
        int smackoverARpop = 2232;

        birth(smackoverARpop) ;
        System.out.println(smackoverARpop);
    }

    static void birth(int cityPop) {
        cityPop++;
    }
}
```

[Listing 10.10](#): Dieses Programm funktioniert nicht.

Wenn Sie das Programm in [Listing 10.10](#) ausführen, zeigt es auf dem Bildschirm die Zahl 2232 an. Nach neun Monaten Planung und Erwartung und sieben langen Stunden im Kreißsaal wurde das Baby der Kermongoos nicht im System registriert. Wie peinlich!

Das Problem ist dadurch hervorgerufen worden, dass Parameter unsauber übergeben worden sind. Wenn Sie in Java einen Parameter, der einen der acht primitiven Typen enthält, an eine Methode übergeben, wird dieser Parameter *als Wert* übergeben. ([Kapitel 4](#) enthält eine Übersicht über die acht primitiven Java-Typen.)

In Klarschrift bedeutet dies, dass jede Änderung, die die Methode am Wert ihres Parameter vornimmt, keinen Einfluss auf die Werte in Variablen hat, die aus dem aufrufenden Code stammen. In [Listing 10.10](#) kann die Methode `birth` den Operator `++` so oft auf `cityPop` anwenden, wie sie will – diese Aktion wirkt sich in keinsten Weise auf den Wert von `smackoverARpop` in der Methode `main` aus.

Technisch gesehen haben wir es hier mit dem Kopieren eines Wertes zu tun (siehe [Abbildung 10.11](#)). Wenn die Methode `main` die Methode `birth` aufruft, wird der Wert von `smackoverARpop` an einen anderen Ort im Arbeitsspeicher kopiert – dem Ort, der für den Wert des Parameters von `cityPop` reserviert ist. Während der Ausführung der Methode `birth` wird der Parameter von `cityPop` um 1 erhöht. Dies geschieht aber nicht dort, wo der ursprüngliche Wert 2232 gespeichert wurde – der Position im Arbeitsspeicher, an der die Variable `smackoverARpop` liegt.

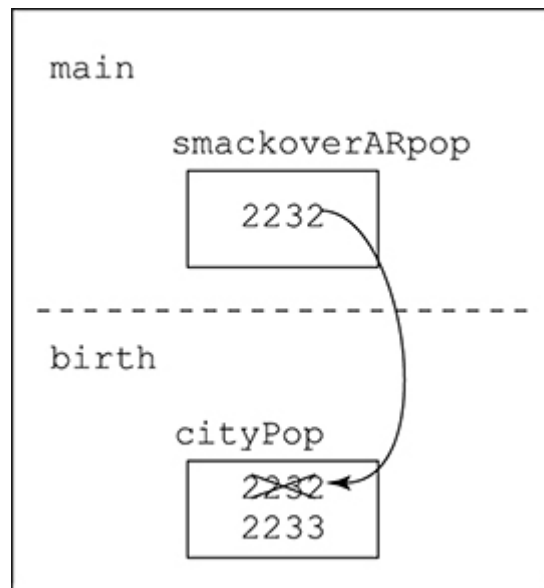


Abbildung 10.11: Übergabe als Wert



Wenn Sie Parameter übergeben, die von einem der acht primitiven Typen sind, werden diese Parameter als Werte übergeben. Diese Werte, die in der Variablen des aufrufenden Codes gespeichert sind, werden dabei nicht geändert. Dies geschieht so selbst dann, wenn die Variable des aufrufenden Codes und der Parameter der aufgerufenen Methode zufällig denselben Namen haben.

Ein Ergebnis zurückgeben

Sie müssen das Problem in [Listing 10.10](#) beheben. Schließlich kann das Baby der Kermongoos nicht unregistriert durchs Leben gehen. Um die Existenz des Babys zu erfassen, müssen Sie den Wert der Variablen `smackoverARpop` um 1 erhöhen. Um dieses Ziel zu erreichen, stehen Ihnen viele Wege offen, von denen der Weg, den [Listing 10.11](#) aufzeigt, nicht der einfachste ist. Aber er hebt einen wichtigen Punkt hervor: Die Rückgabe eines Wertes durch einen Methodenaufruf kann eine mehr als brauchbare Alternative zur Parameterübergabe sein. Schauen Sie sich [Listing 10.11](#) an, um zu sehen, was ich meine.

```
public class TrackPopulation2 {
    public static void main(String args[]) {
```

```

    int smackoverARpop = 2232;

    smackoverARpop = birth(smackoverARpop);
    System.out.println(smackoverARpop);
}

static int birth(int cityPop) {
    return cityPop + 1;
}
}

```

[Listing 10.11](#): Dieses Programm funktioniert.

Wenn Sie den Code in [Listing 10.11](#) ausgeführt haben, ist die Zahl, die Sie dann auf dem Bildschirm Ihres Computers sehen, richtig – 2233.

Im Code von [Listing 10.11](#) gibt es keine neuen Funktionen (außer Sie nennen das *fehlerfreie* Laufen eines Codes eine neue Funktion). Der wichtigste Gedanke, der hinter [Listing 10.11](#) steckt, ist der Befehl `return`, der als solcher bereits in [Kapitel 7](#) zur Sprache kommt. Auf jeden Fall bietet [Listing 10.11](#) eine gute Alternative zur Vorgehensweise in [Listing 10.10](#), auf die Sie besser verzichten sollten.

Übergabe per Referenz

In den letzten ein, zwei Abschnitten habe ich mir große Mühe damit gegeben, auf einen bestimmten Punkt hinzuweisen: Wenn ein Parameter von einem der acht primitiven Typen ist, wird er als Wert übergeben. Vielleicht haben Sie bisher einfach überlesen, dass diese Art der Übergabe nur für primitive Typen gilt. Diese Betonung ist wichtig, weil die Übergabe von Objekten (*Referenztypen*) ganz anders verläuft.

Wenn Sie ein Objekt an eine Methode übergeben, erfolgt diese Übergabe an die andere Methode *per Referenz* (über einen Verweis). Für Sie bedeutet dies, dass Anweisungen in der aufgerufenen Methode Werte ändern *können*, die in den Variablen des Objekts gespeichert sind. Diese Änderungen *wirken sich* auf die Werte *aus*, die der Code sieht, der die Methode aufruft. Die [Listings 10.12](#) und [10.13](#) verdeutlichen diese Situation.

```
class City {
    int population;
}
```

[Listing 10.12](#): Was ist eine City?

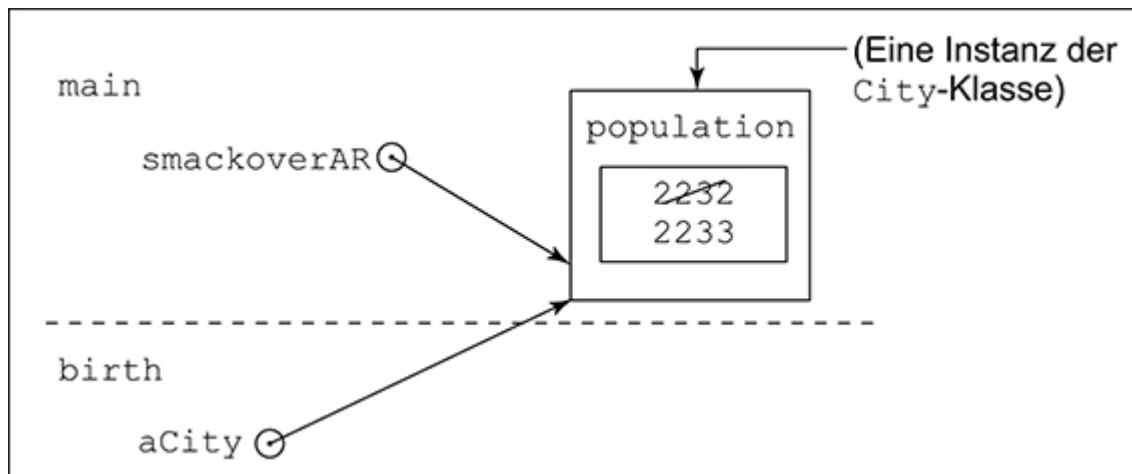
```
public class TrackPopulation3 {
    public static void main(String args[]) {
        City smackoverAR = new City();
        smackoverAR.population = 2232;
        birth(smackoverAR);
        System.out.println(smackoverAR.population);
    }

    static void birth(City aCity) {
        aCity.population++;
    }
}
```

[Listing 10.13](#): Einer Methode ein Objekt übergeben

Wenn Sie den Code der [Listings 10.12](#) und [10.13](#) ausführen, wird als Ergebnis 2233 ausgegeben. Das ist auch richtig so, denn im Code gibt es Dinge wie ++ und birth. Durch das Hinzufügen von 1 zu aCity.population innerhalb der Methode birth wird nun auch der Wert von smackoverAR.population in der Methode main tatsächlich geändert.

Wie die Methode birth den Wert von smackoverAR.population ändert, stellt [Abbildung 10.12](#) schematisch dar. Wenn Sie ein Objekt an eine Methode übergeben, legt der Computer keine vollständige Kopie des Objekts an. Stattdessen wird nur eine Kopie der Referenz auf das Objekt erzeugt. (Stellen Sie sich den Weg in [Abbildung 10.12](#) so vor, als wenn der Computer einen Pfeil kopiert, der auf das Objekt zeigt.)



[Abbildung 10.12](#): Übergabe per Referenz

Sie sehen in [Abbildung 10.12](#) nur eine Instanz der Klasse `City`, die eine Variable `population` enthält.

Behalten Sie dieses Objekt im Auge, während Sie die folgenden Schritte durchlesen:

1. Unmittelbar bevor die Methode `birth` aufgerufen wird, verweist die Variable `smackoverAR` auf dieses Objekt – die Instanz der Klasse `City`.
2. Wenn die Methode `birth` aufgerufen wird und die Variable `smackoverAR` an den Parameter der Methode `aCity` übergeben worden ist, kopiert der Computer die Referenz von `smackoverAR` nach `aCity`. Nun verweist `aCity` auf dasselbe Objekt – die Instanz der Klasse `City`.
3. Wenn der Befehl `aCity.population++` in der Methode `birth` ausgeführt wird, erhöht der Computer das Feld `population` des Objekts um 1. Nun wird im Feld `population` der einzigen Instanz von `City` die Einwohnerzahl 2233 gespeichert.
4. Bei der Programmausführung wieder jetzt wieder die Methode `main` angesprochen. Der Wert von `smackoverAR.population` wird ausgegeben. Da `smackoverAR` noch auf die Instanz der Klasse `City` verweist, hat `smackoverAR.population` den Wert 2233. Die Familie Kermongoos ist so stolz auf ihren Nachwuchs.

Ein Objekt von einer Methode zurückgeben

Ob Sie es glauben oder nicht, aber trotz der Informationen über die Übergabe von Parametern in den letzten Abschnitten gibt es in Java immer noch einen Winkel, in dem Unentdecktes schlummert, das mit Methoden zu tun hat. Wenn Sie eine Methode aufrufen, kann sie etwas unmittelbar an den aufrufenden Code übergeben. Ich gehe in früheren Kapiteln und Abschnitten zu primitiven Werten (wie zum Beispiel `int`-Werte) darauf ein. Alternativ kann sie aber auch nichts (auch als `void` bekannt) zurückgeben. In diesem Abschnitt gebe ich ein vollständiges Objekt zurück. Es handelt sich dabei um ein Objekt vom Typ `City` aus [Listing 10.12](#). Der Code, der zu so etwas fähig ist, befindet sich in [Listing 10.14](#).

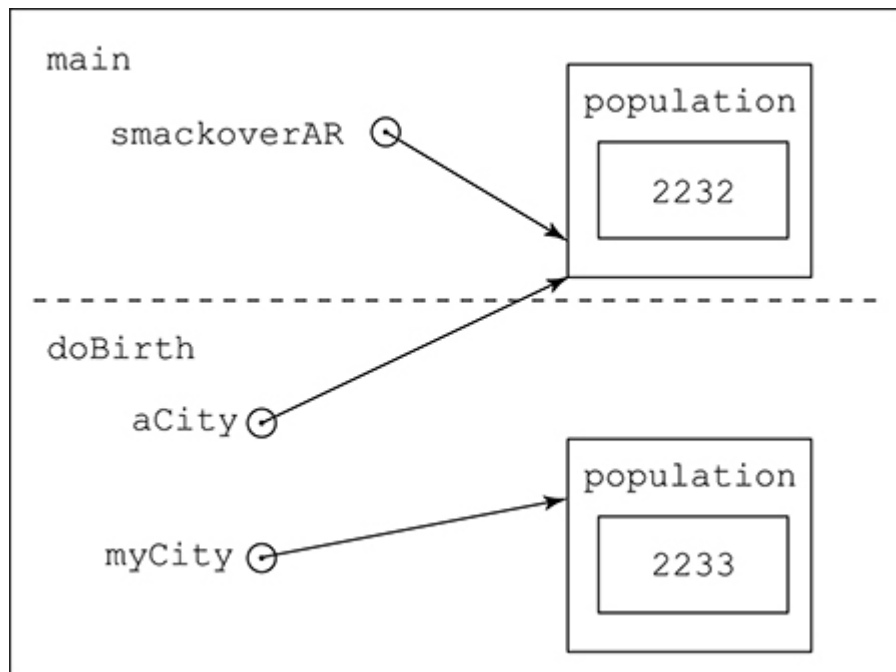
```
public class TrackPopulation4 {

    public static void main(String args[]) {
        City smackoverAR = new City();
        smackoverAR.population = 2232;
        smackoverAR = doBirth(smackoverAR);
        System.out.println(smackoverAR.population);
    }

    static City doBirth(City aCity) {
        City myCity = new City();
        myCity.population = aCity.population + 1;
        return myCity;
    }
}
```

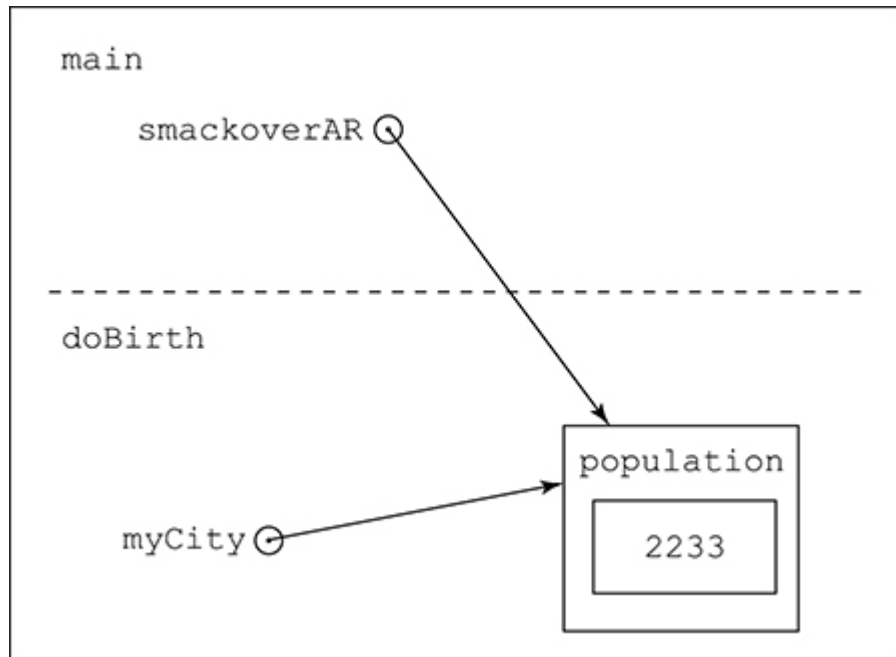
[Listing 10.14](#): Hier, eine `City` für Sie.

Wenn Sie den Code aus [Listing 10.14](#) ablaufen lassen, erhalten Sie die Zahl 2233. Das geht in Ordnung. Der Code weist die Methode `doBirth` an, eine neue Instanz von `City` anzulegen. In dieser Instanz hat `population` den Wert 2233 (siehe [Abbildung 10.13](#)).



[Abbildung 10.13](#): Die Methode *doBirth* erstellt eine Instanz von *City*.

Wenn die Methode `doBirth` ausgeführt worden ist, wird die Instanz von `City` an die Methode `main` zurückgegeben. Dann wird diese zurückgegebene Instanz der Variablen `smackoverAR` zugewiesen (siehe [Abbildung 10.14](#)). Nun verweist `smackoverAR` auf eine funkelneue Instanz von `City` – eine Instanz, deren Einwohnerzahl (*Population*) 2233 beträgt.



[Abbildung 10.14](#): Die neue Instanz von *City* wird der Variablen *smackoverAR* zugewiesen.

Achten Sie in [Listing 10.14](#) beim Aufruf und bei der Rückgabe der Methode `doBirth` auf die Konsistenz der Typen:

- ✓ Die Variable `smackoverAR` ist vom Typ `City`. Diese Variable wird an den Parameter `aCity` übergeben, dessen Typ ebenfalls `City` ist.
- ✓ Die Variable `myCity` ist vom Typ `City`. Diese Variable wird in der Rückgabeeinweisung der Methode `doBirth` zurückgesendet. Auch das ist konsistent, denn der Kopf der Methode `doBirth` beginnt mit `static City doBirth(bla, bla, bla ...` – der Zusage, ein Objekt vom Typ `City` zurückzugeben.
- ✓ Die Methode `doBirth` gibt ein Objekt vom Typ `City` zurück. In der Methode `main` wird dann das Objekt, das der Aufruf von `doBirth` zurückgegeben hat, der Variablen `smackoverAR` zugewiesen, die (Sie haben es richtig erraten) auch vom Typ `City` ist.

Diese Übereinstimmung der Typen wirkt nicht nur sehr harmonisch, sondern sie ist auch absolut notwendig. Wenn Sie

ein Programm schreiben, in dem Ihre Typen nicht übereinstimmen, wirft der Compiler eine unsympathische Meldung aus, die von inkompatiblen Typen handelt.

Nachtrag

Dora Kermongoos und ihre neugeborene Tochter leben gesund und munter in ihrem Heim in Smackover, Arkansas.

11

Arrays verwenden, um mit Werten zu jonglieren

In diesem Kapitel

- ▶ Mit mehreren Werten auf einmal umgehen
 - ▶ Dinge suchen
 - ▶ Werte zu Beginn der Programmausführung erstellen
-

Willkommen im Java-Motel! Keine arroganten Pagen, kein überteuerter Zimmerservice, keine dummen Sprüche. Einfach nur ein sauberes Doppelzimmer zu einem ausgesprochen günstigen Preis!

Bereiten Sie sich gut vor

Das Java-Motel hat zehn komfortable Zimmer und befindet sich in einer ruhigen Gegend. Mit Ausnahme eines kleinen, separaten Büros besteht das Motel aus einer langen Reihe von Zimmern. Jedes Zimmer kann vom großzügigen Parkplatz aus betreten werden.

Es sieht ziemlich seltsam aus, aber die Zimmer sind von 0 bis 9 durchnummeriert. Ich könnte nun behaupten, dass das Zufall sei – etwas, das mit den Plänen des Architekten zu tun hatte, aber wenn die Nummerierung der Zimmer mit 0 anfängt, lassen sich die Beispiele in diesem Kapitel leichter verstehen.

Auf jeden Fall müssen Sie immer wissen, wie viele Gäste sich in den einzelnen Zimmern aufhalten. Da es zehn Zimmer gibt, könnten Sie der Meinung sein, dass die Deklaration von zehn Variablen die beste Lösung sei:

```
int guestsInRoomNum0, guestsInRoomNum1, guestsInRoomNum2,
    guestsInRoomNum3, guestsInRoomNum4, guestsInRoomNum5,
    guestsInRoomNum6, guestsInRoomNum7, guestsInRoomNum8,
    guestsInRoomNum9;
```

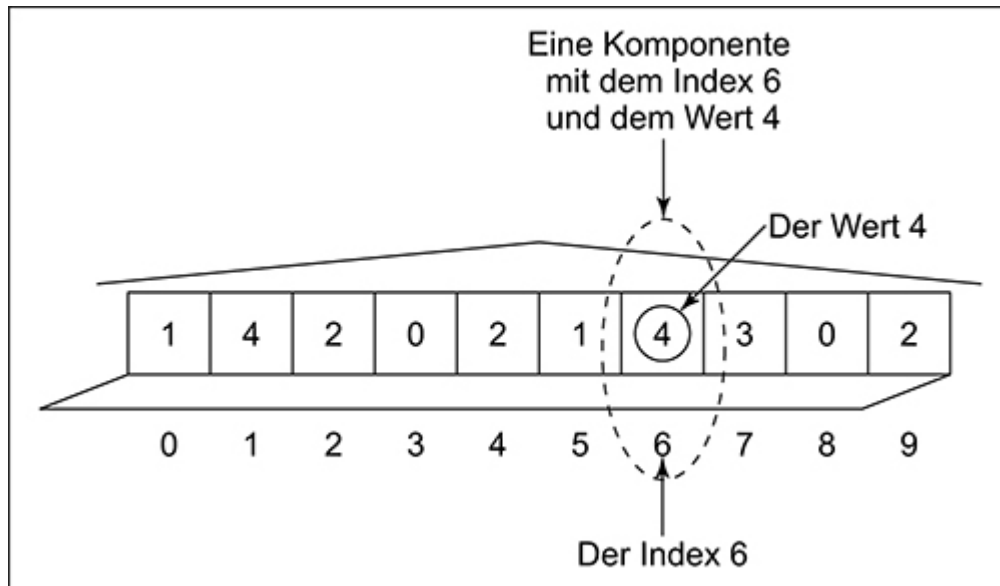
Dieser Ansatz ist ziemlich ineffizient – aber Ineffizienz ist nicht das Einzige, was an diesem Code zu kritisieren ist. Noch problematischer ist, dass Sie diese Variablen nicht in einer Schleife durchlaufen können. Wenn Sie den Wert einer jeden Variablen auslesen wollen, müssen Sie die Methode `nextInt` zehnmal kopieren:

```
guestsInRoomNum0 = diskScanner.nextInt();
guestsInRoomNum1 = diskScanner.nextInt();
guestsInRoomNum2 = diskScanner.nextInt();
... und so weiter.
```

Aber es gibt einen besseren Weg.

Zu diesem Weg gehört ein Array. Ein *Array* besteht aus einer Reihe von Werten – so wie die aneinandergereihten Zimmer eines Motels, das nur aus einem Erdgeschoss besteht. Wenn Sie sich ein Bild von einem Array machen wollen, vergleichen Sie es einfach mit dem Java-Motel:

- ✓ Stellen Sie sich als Erstes die Zimmer vor, die nebeneinander aufgereiht sind.
- ✓ Stellen Sie sich dann dieselben Zimmer mit fehlenden Außenwänden vor. In den Zimmern sehen Sie eine bestimmte Anzahl an Gästen.
- ✓ Versuchen Sie, die beiden Gäste in Zimmer 9 zu vergessen, die Banknoten in einem Aktenkoffer stapeln. Ignorieren Sie die Gäste in Zimmer 6, die seit eineinhalb Tagen vor dem Fernsehgerät abhängen. Konzentrieren Sie sich bei all diesen Details nur auf Zahlen. In jedem Zimmer gibt eine Zahl wieder, wie viele Gäste es beherbergt. (Wenn Ihre Stärke nicht das Abstrahieren ist, schauen Sie sich [Abbildung 11.1](#) an.)



[Abbildung 11.1](#): Ein abstrakter Schnappschuss der Zimmer des Java-Motels

In der Terminologie dieses Kapitels wird eine ganze Zimmerzeile *Array* genannt. Jedes Zimmer dieses Arrays ist eine *Komponente* des Arrays (was auch als *Element* des Arrays bezeichnet wird). Jeder Komponente sind zwei Zahlen zugeordnet:

- ✓ Die Zimmernummer (eine Zahl von 0 bis 9), die auch als *Index* des Arrays bezeichnet wird.
- ✓ Eine Zahl von Gästen; dabei handelt es sich um einen *Wert*, der in einer Komponente des Arrays gespeichert wird.

Der Einsatz eines Arrays erspart Ihnen diesen Wiederholungsunsinn des Beispielcodes vom Anfang dieses Kapitels. Wenn Sie zum Beispiel ein Array mit zehn Werten deklarieren wollen, reicht ein kurzer Befehl aus:

```
int guests[] = new int[10];
```

Wenn Sie es sehr ausführlich lieben, können Sie diese Anweisung so erweitern, dass sie aus zwei separaten Befehlen besteht:

```
int guests[];  
guests = new int[10];
```

Beachten Sie bei beiden Codestückchen die Verwendung der Zahl 10. Sie weist den Computer an, das Array `guests` (deutsch *Gäste*) so einzurichten, dass es aus zehn Komponenten besteht.

Jede Komponente eines Arrays hat einen eigenen Namen. Die Komponente, mit der es anfängt, heißt `guests[0]`, die nächste `guests[1]` und so weiter. Die letzte der zehn Komponenten trägt den Namen `guests[9]`.



Wenn Sie ein Array erstellen, geben Sie immer an, wie viele Komponenten es enthält. Die Indexe eines Arrays (die im Deutschen manchmal auch *Indizes* genannt werden) beginnen mit 0 und hören mit der Zahl auf, die um eins kleiner ist als die Gesamtzahl der Komponenten.

Die Codestückchen, die ich Ihnen zeige, stellen zwei Wege dar, um ein Array zu erstellen. Das erste Beispiel kommt mit einer Zeile aus. Das zweite verwendet zwei Zeilen. Wenn Sie einzeilig weitermachen, können Sie diese eine Zeile innerhalb oder außerhalb einer Methode schreiben. Sie haben die freie Wahl. Wenn Sie aber zwei separate Zeilen verwenden, sollte die zweite Zeile – in diesem Fall `guests = new int[10]` – immer innerhalb einer Methode stehen.



Sie können bei der Deklaration eines Arrays die eckigen Klammern vor oder hinter den Namen der Variablen setzen. Sie können also entweder `int guests[]` oder `int[] guests` schreiben. Der Computer erstellt unabhängig davon, wie Sie vorgehen, immer dieselbe Variable `guests`.

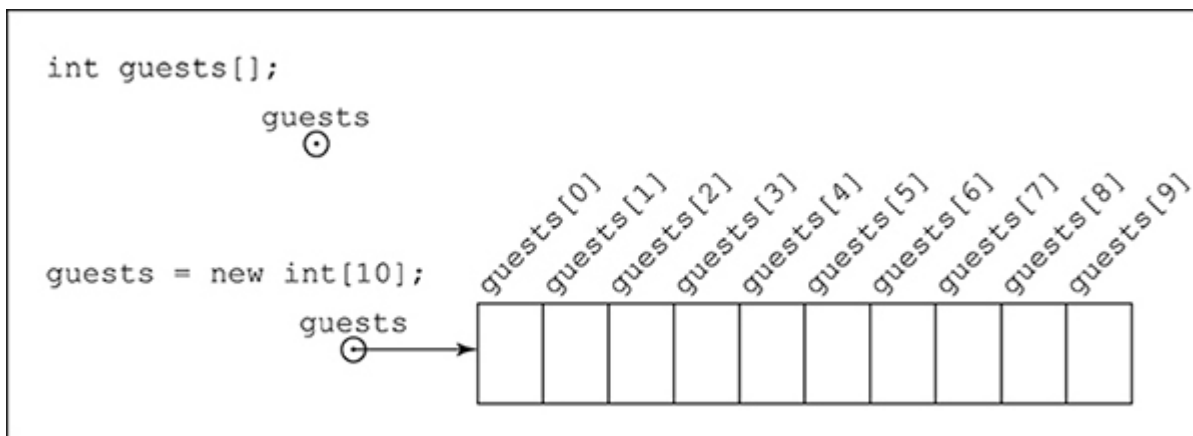
Ein Array in zwei einfachen Schritten erstellen

Schauen Sie sich noch einmal die beiden Zeilen an, die Sie verwenden können, um ein Array zu erstellen:

```
int guests[];  
guests = new int[10];
```

Jede Zeile erfüllt einen ganz bestimmten Zweck:

- ✓ **int guests[]:** Bei dieser ersten Zeile handelt es sich um eine Deklaration. Diese Deklaration reserviert den Namen des Arrays (einen Namen wie `guests`), damit er im restlichen Programm verwendet werden kann. Im Bild mit dem Java-Motel sagt diese Zeile: »Ich plane hier den Bau eines Motels und bringe in jedem Zimmer eine bestimmte Anzahl an Gästen unter.« (Siehe [Abbildung 11.2.](#))



[Abbildung 11.2:](#) Ein Array wird in zwei Schritten erstellt.

Kümmern Sie sich nicht darum, was die Deklaration `int guests[]` tatsächlich macht. Es ist viel wichtiger, dass Sie darauf achten, was die Deklaration `int guests[]` *nicht* macht. Sie reserviert keine zehn Stellen im Arbeitsspeicher. Letztendlich ist es sogar so, dass eine Deklaration wie `int guests[]` noch nicht einmal ein Array erstellt. Diese Deklaration macht nichts, als die Variable `guests` einzurichten. An dieser Stelle im Code verweist die Variable `guests` nicht auf ein echtes Array. (Oder anders ausgedrückt: Das Motel hat zwar schon einen Namen, aber es ist noch nicht gebaut worden.)

- ✓ **guests = new int[10]:** Bei der zweiten Zeile handelt es sich um einen Zuweisungsbefehl. Dieser Zuweisungsbefehl reserviert für zehn `int`-Werte Platz im Arbeitsspeicher des Computers. In der Terminologie der Grundbesitzer besagt dies: »Ich habe endlich das Motel gebaut. Jetzt können wir

anfangen, die Zimmer mit Gästen zu belegen.« (Siehe auch hierzu [Abbildung 11.2](#).)

Werte speichern

Nachdem Sie nun ein Array erstellt haben, können Sie in seinen Komponenten Werte ablegen. So möchten Sie zum Beispiel die Tatsache speichern, dass es in Zimmer 6 4 Gäste gibt. Um in der Komponente mit dem Index 6 den Wert 4 zu hinterlegen, schreiben Sie `guests[6] = 4`.

Und jetzt fängt das Motel an zu laufen. Ein großer Bus fährt vor. Er trägt die Aufschrift »Noahs Arche«. 25 Paare verlassen den Bus und bewegen sich gehend, stampfend, fliegend, hoppelnd oder kriechend zum kleinen Büro des Motels. Im Java-Motel können nur 10 Paar bleiben, was aber in Ordnung ist, denn Sie können die verbliebenen 15 Paare zur alten *C-Side Resort and Motor Lodge* schicken, die nicht weit entfernt ist.

Um nun die 10 Paare zu registrieren, bringen Sie in jedem der zehn Zimmer des Java-Motels ein Paar (2 Gäste) unter. Wenn Sie erst einmal ein Array erstellt haben, können Sie sich dessen Indexierungsmöglichkeit zunutze machen und eine `for`-Schleife wie diese hier schreiben:

```
for (int roomNum = 0; roomNum < 10; roomNum++) {  
    guests[roomNum] = 2;  
}
```

Diese Schleife ersetzt zehn Zuweisungsbefehle. Beachten Sie, dass der Schleifenzähler von 0 bis 9 geht. Vergleichen Sie dies mit [Abbildung 11.2](#) und denken Sie daran, dass die Indexe eines Arrays mit 0 beginnen und mit einem Wert aufhören, der um eins kleiner ist als die Gesamtzahl der Komponenten des Arrays.

Leider geht es in der Welt nicht nur geordnet zu: Ihre Gäste kommen nicht immer paarweise an, und Sie müssen in jedem Zimmer eine andere Zahl von Gästen unterbringen. Sie könnten die Informationen über Gäste und die Zimmerbelegung in einer Datenbank ablegen. In diesem Fall können Sie das Array immer noch in einer Schleife durchlaufen und dabei die Zahl der

jeweiligen Gäste abfragen. Der Code, mit dem sich eine solche Aufgabe erledigen lässt, könnte so aussehen:

```
resultset =
    statement.executeQuery("select GUESTS from RoomData
");
for (int roomNum = 0; roomNum < 10; roomNum++) {
    resultset.next();
    guests[roomNum] = resultset.getInt("GUESTS");
}
```

Da in diesem Buch aber erst in [Kapitel 17](#) auf Datenbanken eingegangen wird, ist es im Moment besser für Sie, die Zahl der Gäste aus einer Textdatei zu lesen. [Abbildung 11.3](#) zeigt als Beispiel die Datei `GuestList.txt`.

1 4 2 0 2 1 4 3 0 2

[Abbildung 11.3](#): Die Datei *GuestList.txt*

Wenn Sie die Datei angelegt haben, können Sie die Klasse `Scanner` aufrufen, um Werte aus der Datei auszulesen. Der entsprechende Code steht in [Listing 11.1](#), und das, was dieser Code ausgibt, wird in [Abbildung 11.4](#) wiedergegeben.

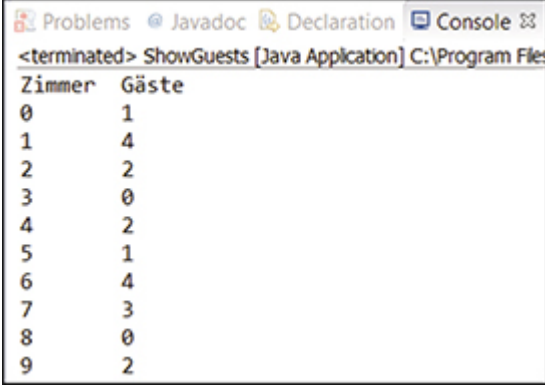
```
import static java.lang.System.out;
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
public class ShowGuests {
    public static void main(String args[])
                                throws IOException
    {
        int guests[] = new int[10];
        Scanner diskScanner =
            new Scanner(new File("GuestList.txt"))
;
    }
```

```

        for(int roomNum = 0; roomNum < 10; roomNum++)
    {
        guests[roomNum] = diskScanner.nextInt();
    }
    out.println("ZimmertGäste");
    for(int roomNum = 0; roomNum < 10; roomNum++)
    {
        out.print(roomNum);
        out.print("t");
        out.println(guests[roomNum]);
    }
    diskScanner.close();
}
}

```

[Listing 11.1](#): Ein Array mit Werten füllen



Zimmer	Gäste
0	1
1	4
2	2
3	0
4	2
5	1
6	4
7	3
8	0
9	2

[Abbildung 11.4](#): Das Programm in [Listing 11.1](#) wird ausgeführt.

Im Code von [Listing 11.1](#) gibt es zwei `for`-Schleifen. Die erste Schleife liest die Zahl der Gäste aus, während die zweite Schleife Gästezahlen schreibt.



Jedes Array besitzt ein internes Längengebiet. Die *Länge* eines Arrays entspricht der Anzahl der Komponenten im Array. Wenn Sie in [Listing 11.1](#) den Wert von `guests.length` ausgeben, erhalten Sie 10.

Tabulatoren und anderes

In [Listing 11.1](#) verwenden einige Aufrufe von `print` und `println` die Escape-Folge `\t`. Sie wird *Escape-Folge* genannt, weil Sie der Anzeige des Buchstabens `t` auf dem Bildschirm »entkommen« (*to escape* bedeutet im Deutschen *entkommen*). Die Buchstaben `\t` stehen für einen Tabulator. Der Computer geht an die Position des nächsten Tabulators, bevor weitere Zeichen ausgegeben werden. Java kennt eine Reihe dieser praktischen Escape-Folgen, von denen einige in [Tabelle 11.1](#) abgebildet sind.

b	Rückschritt (Backspace)
t	Horizontaler Tabulator
n	Neue Zeile (Line Feed)
f	Neue Seite (Form Feed)
r	Wagenrücklauf (Carriage Return)
"	Doppelte Anführungszeichen (")
'	Einfaches Anführungszeichen (')
	Umgekehrter Schrägstrich (Backslash \)

[Tabelle 11.1](#): Escape-Folgen

Einen Array-Initialisierer verwenden

Zusätzlich zu dem, was [Listing 11.1](#) zeigt, haben Sie noch eine andere Möglichkeit, um in Java ein Array zu füllen – einen *Array-Initialisierer*. Wenn Sie einen Array-Initialisierer verwenden, müssen Sie dem Computer noch nicht einmal mitteilen, aus wie vielen Komponenten das Array besteht. Der Computer findet dies dann selbstständig heraus.

[Listing 11.2](#) enthält eine neue Version des Codes, um ein Array zu füllen. Das Programm gibt dasselbe aus wie das Programm in [Listing 11.1](#). (Es handelt sich um die Daten, die [Abbildung 11.4](#) zeigt.) Fettschrift hebt in [Listing 11.2](#) die Unterschiede zwischen den [Listings 11.1](#) und [11.2](#) hervor. Das fett Gedruckte stellt einen Array-Initialisierer dar.

```
import static java.lang.System.out;
public class ShowGuests {
```

```

    public static void main(String args[]) {

        int guests[] = {1, 4, 2, 0, 2, 1, 4, 3, 0, 2}
;
        out.println("ZimmertGäste");

        for (int roomNum = 0; roomNum < 10; roomNum++)
    ) {
            out.print(roomNum);
            out.print("t");
            out.println(guests[roomNum]);
        }
    }
}

```

[Listing 11.2](#): Einen Array-Initialisierer verwenden

}



Ein Array-Initialisierer kann sowohl Ausdrücke als auch Literale enthalten. Dies bedeutet in Klarschrift, dass Sie zwischen die Kommata im Initialisierer alles Mögliche packen können. So funktioniert zum Beispiel auch ein Initialisierer wie

```
{1 + 3, keyboard.nextInt(), 2, 0, 2, 1, 4, 3, 0, 2}
```

richtig gut.

Ein Array mit einer erweiterten »for«-Schleife durchlaufen

Java kennt eine erweiterte `for`-Schleife, die ohne Zähler oder Indexe auskommt. [Listing 11.3](#) zeigt Ihnen, wie das geht.



Dieses Kapitel verlangt Java 5.0 oder später. Die Codes laufen nicht mit älteren Java-Versionen wie Java 1.3 oder 1.5. Sie finden in [Kapitel 2](#) Informationen über die Versionsnummerierung von Java.

```

import static java.lang.System.out;
public class ShowGuests {
    public static void main(String args[]) {

```

```

        int guests[] = {1, 4, 2, 0, 2, 1, 4, 3, 0, 2}
;
        int roomNum = 0;
        out.println("ZimmertGäste");
        for (int numGuests : guests) {
            out.print(roomNum++);
            out.print("t");
            out.println(numGuests);
        }
    }
}

```

[Listing 11.3](#): Ein Array mit `for`-Schleife

Die [Listings 11.1](#) und [11.3](#) geben dasselbe aus (siehe weiter vorn im Kapitel [Abbildung 11.4](#)).

Wenn Sie sich die Schleife in [Listing 11.3](#) anschauen, sehen Sie ein altbekanntes Muster. Die Schleife dieses Beispiels besteht wie die Schleifen in Listing 6.5 aus drei Teilen:

`for (Variablentyp Variablenname : Wertebereich)`

Die ersten beiden Teile sind *Variablentyp* und *Variablenname*. Die Schleife in [Listing 11.3](#) definiert den Variablennamen `numGuests`, und `numGuests` ist vom Typ `int`. Die Variable `numGuests` nimmt bei jedem Schleifendurchlauf einen neuen Wert an. [Abbildung 11.4](#) gibt diese Werte wieder. Der Initialisierungswert ist 1. Der nächste Wert ist 4, danach kommt 2. Und so weiter.

Woher holt die Schleife diese Zahlen? Die Antwort liegt im *Wertebereich* der Schleife. In [Listing 11.3](#) besteht der *Wertebereich* der Schleife aus `guests`. Das bedeutet, dass beim ersten Durchlauf der Schleife ihr Initialisierungswert `guests[0]` ist (was 1 bedeutet). Beim nächsten Durchlauf beträgt der Wert von `numGuests` `guests[1]` (entspricht 4). Danach kommt `guests[2]` (beträgt 2). Und so weiter.



Javas erweiterte `for`-Schleife verlangt einen warnenden Hinweis. Bei jedem Schleifendurchlauf speichert die Variable,

die im Wertebereich unterwegs ist, eine *Kopie* des Wertes. Sie zeigt also nicht auf den Wertebereich selbst.

Wenn Sie zum Beispiel einen Zuweisungsbefehl hinzufügen, der in [Listing 11.3](#) den Wert von `numGuests` ändert, wirkt sich dieser Befehl auf keinen der Werte aus, die im Array `guests` gespeichert worden sind. Um diesen Punkt zu schließen, sollten Sie sich vorstellen, dass die Geschäfte schlecht gehen und das Array `guests` mit Nullen gefüllt ist. Dann führe ich den folgenden Code aus:

```
for (int numGuests : guests) {  
    numGuests += 1;  
    out.print(numGuests + " ");  
}  
out.println();  
for (int numGuests : guests) {  
    out.print(numGuests + " ");  
}
```

Die Variable `numGuests` nimmt Werte an, die im Array `guests` gespeichert sind. Aber die Anweisung `numGuests += 1` ändert die im Array gespeicherten Werte nicht. Das, was der Code ausgibt, sieht dann so aus:

```
1 1 1 1 1 1 1 1 1 1  
0 0 0 0 0 0 0 0 0 0
```

Suchen

Sie sitzen an der Rezeption des Java-Motels. Aufgewacht! Da kommt eine Gruppe, die aus fünf Personen besteht. Diese Leute möchten ein Zimmer, weshalb Sie eine Software benötigen, die herausfindet, ob ein Zimmer frei ist. In diesem Fall soll die Software die Datei `GuestList.txt` (siehe [Abbildung 11.3](#)) ändern und die Zahl 0 durch die Zahl 5 ersetzen. Sie haben Glück, denn die entsprechende Software – sie wird in [Listing 11.4](#) gezeigt – befindet sich auf Ihrer Festplatte.

```
import static java.lang.System.out;  
import java.util.Scanner;  
import java.io.File;  
import java.io.IOException;
```



```

import java.io.PrintStream;
public class FindVacancy {

    public static void main(String args[])
                                throws IOException
    {

        int guests[] = new int[10];
        int roomNum;

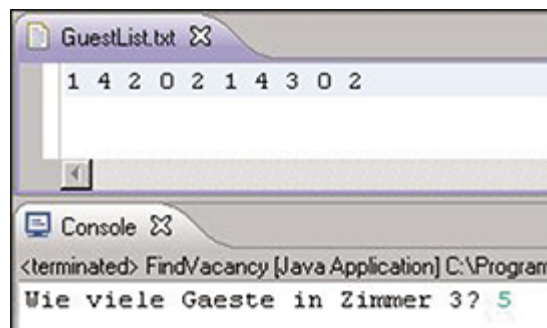
        Scanner diskScanner =
            new Scanner(new File("GuestList.txt"));
        for (roomNum = 0; roomNum < 10; roomNum++) {
            guests[roomNum] = diskScanner.nextInt();
        }
        diskScanner.close();
        roomNum = 0;
        while (roomNum < 10 && guests[roomNum] != 0)
    {
        roomNum++;
    }
    if (roomNum == 10) {
        out.println("Leider ist kein Zimmer frei.
");
    } else {
        out.print("Wie viele Gäste in Zimmer ");
        out.print(roomNum);
        out.print("? ");
        Scanner keyboard = new Scanner(System.in)
;
        guests[roomNum] = keyboard.nextInt();
        keyboard.close();

        PrintStream listOut =
            new PrintStream("GuestList.txt");
        for (roomNum = 0; roomNum < 10; roomNum++)
    {
        listOut.print(guests[roomNum]);
        listOut.print(" ");
    }
    listOut.close();
    }
}

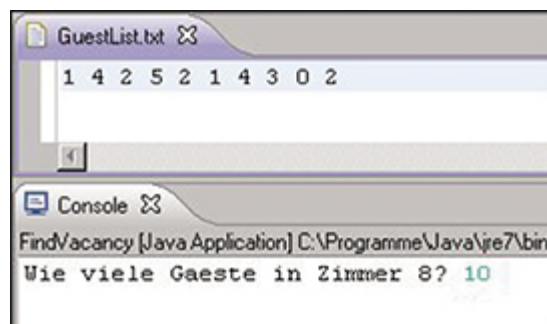
```

[Listing 11.4](#): Haben Sie ein freies Zimmer?

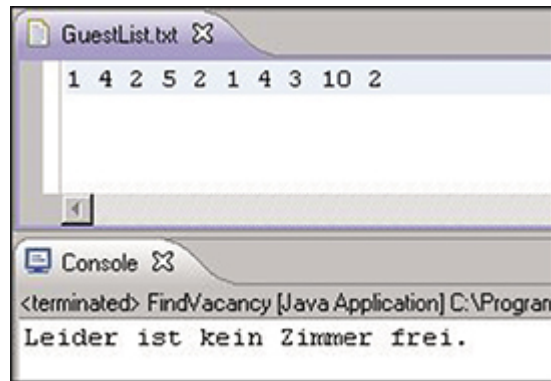
[Abbildung 11.5](#), [Abbildung 11.6](#) und [Abbildung 11.7](#) stellen dar, wie der Code in [Listing 11.4](#) ausgeführt wird. Wie [Abbildung 11.3](#) noch zeigt, gibt es im Motel zwei freie Zimmer – die Zimmer 3 und 8. (Denken Sie daran, dass die Nummerierung der Zimmer mit 0 beginnt.) Wenn Sie den Code in [Listing 11.4](#) zum ersten Mal ausführen, erfahren Sie, dass Zimmer 3 frei ist, und das Programm bringt in diesem Zimmer die 5 Personen unter. Wenn Sie den Code zum zweiten Mal ausführen, findet das Programm das andere leere Zimmer (Zimmer 8) und bringt dort eine Gruppe von 10 Personen unter. Und beim dritten Ausführen des Codes gibt es kein freies Zimmer mehr. Wenn das Programm dies entdeckt, gibt es die Meldung
Leider ist kein Zimmer frei aus.



[Abbildung 11.5](#): Ein freies Zimmer belegen



[Abbildung 11.6](#): Das letzte freie Zimmer belegen



[Abbildung 11.7](#): Leider sind alle Zimmer belegt.



Dadurch, dass Sie den Code in [Listing 11.4](#) ausführen, wird eine ganz neue Datei `GuestList.txt` geschrieben. Dies kann zu Verwirrung führen, weil jede Java-IDE den Inhalt der Datei `GuestList.txt` auf ihre eigene Weise darstellt. Einige IDEs zeigen den Inhalt der neuen Datei nicht automatisch an. Es kann also sein, dass Sie die an der Datei vorgenommenen Änderungen nicht sofort sehen. (Zum Beispiel ist in [Abbildung 11.5](#) Zimmer 3 nicht belegt. Aber nachdem Sie den Code in [Listing 11.4](#) ausgeführt haben, ist Zimmer 3 mit 5 Gästen belegt.) Selbst wenn Sie keine Änderungen feststellen, ändert das wiederholte Ausführen des Codes in [Listing 11.4](#) die Datei `GuestList.txt`. Spielen Sie mit der von Ihnen bevorzugten IDE herum, um herauszufinden, wie Sie es schaffen, dass die IDE die Anzeige der Textdatei aktualisiert.



In [Listing 11.4](#) ist die Bedingung `roomNum < 10 && guests[roomNum] != 0` ziemlich trickreich. Aber wenn Sie den Code etwas umstellen und `guests [roomNum] != 0 && roomNum < 10` schreiben, bekommen Sie ernsthafte Probleme.

In eine Datei schreiben

Der Code in [Listing 11.4](#) verwendet Tricks aus anderen Kapiteln und Abschnitten dieses Buches. Die einzig wirklich neue Funktion

ist der Einsatz von `PrintStream`, um in eine Datei auf der Festplatte zu schreiben. Denken Sie an die Beispiele, die `System.out.print`, `out.println` oder Variationen davon aufrufen. Was passiert eigentlich genau, wenn Sie eine dieser Methoden aufrufen?

Bei dem Teil mit dem Namen `System.out` handelt es sich um ein Objekt. Dieses Objekt wird in der Java-API definiert. Und letztendlich ist `System.out` sogar die Instanz einer Klasse, die `java.io.PrintStream` heißt (und von ihren besten Freunden `PrintStream` genannt wird). Nun verfügt jedes Objekt, das aus der Klasse `PrintStream` erstellt wird, über die Methoden `print` und `println`. Und so, wie jedes `Konto`-Objekt in [Listing 7.3](#) eine Methode `display` und das Objekt `DecimalFormat` aus [Listing 10.1](#) eine Methode `format` enthalten, verfügt das Objekt `out` von `PrintStream` über die Methoden `print` und `println`. Wenn Sie `System.out.println` aufrufen, betrifft dies eine Methode, die zu einer Instanz von `PrintStream` gehört.

Gut, aber was hat das zu bedeuten? Nun, `System.out` steht immer für Textbereiche auf dem Bildschirm Ihres Computers. Wenn Sie ein eigenes `PrintStream`-Objekt erstellen und dafür sorgen, dass dieses Objekt auf eine Datei auf der Festplatte verweist, schreiben Sie Text in diese Datei, wenn Sie die Methode `print` des `PrintStream`-Objekts aufrufen.

Wenn Sie also in [Listing 11.4](#) schreiben

```
PrintStream listOut =  
    new PrintStream("GuestList.txt");  
listOut.print(guests[roomNum]);  
listOut.print(" ");
```

weisen Sie Java an, Text in die Datei auf Ihrer Festplatte – die Datei `GuestList.txt` – zu schreiben.

Auf diese Weise aktualisieren Sie die Gästeliste des Motels. Wenn Sie für die Anzeige der Gäste in Zimmer 3 `listOut.print` aufrufen, geben Sie die Zahl 5 aus. Zwischen [Abbildung 11.5](#) und [Abbildung 11.6](#) muss also mit der Datei `GuestList.txt` etwas geschehen sein, weil sich die Anzahl der

Gäste in Zimmer 3 von 0 in 5 geändert hat. In [Abbildung 11.6](#) wird das Programm zum zweiten Mal ausgeführt. Wenn es dann von der neu geschriebenen Datei `GuestList.txt` Daten erhält, steht Zimmer 3 nicht mehr leer, weshalb nun Zimmer 8 vorgeschlagen wird.



Dies ist mehr eine Beobachtung als ein Tipp. Gehen wir davon aus, dass Sie Daten aus einer Datei mit dem Namen `Mitarbeiter.txt` *lesen* wollen. Um dies zu erledigen, rufen Sie `new Scanner(new File("Mitarbeiter.txt"))` auf. Sollten Sie unglücklicherweise `new Scanner("Mitarbeiter.txt")` ohne den Teil mit `new File` aufrufen, stellt der Aufruf keine Verbindung zur Datei `Mitarbeiter.txt` her. Beachten Sie nun, wie das *Schreiben* von Daten in eine Datei vorbereitet wird. Sie legen eine Instanz von `PrintStream` an, indem Sie `new PrintStream("GuestList.txt")` aufrufen. In diesem Aufruf kommt `new File` nicht vor. Wenn Sie nicht aufpassen und unglücklicherweise `new File` in den Aufruf aufnehmen, wird der Compiler sauer.

Wann eine Datei geschlossen werden muss

Achten Sie in [Listing 11.4](#) auf die Platzierung der Aufrufe von `new Scanner`, `new PrintStream` und `close`. Wie bei allen Beispielen enthält auch der Aufruf von `new Scanner` einen hierzu gehörenden Aufruf von `close`. In [Listing 11.4](#) verfügt der Aufruf von `new PrintStream` über einen eigenen `close`-Aufruf (den Aufruf `listOut.close()`). In [Listing 11.4](#) achte ich peinlich genau darauf, diese Aufrufe direkt bei den Aufrufen von `nextInt` und `print` zu platzieren. So habe ich zum Beispiel `diskScanner` nicht gleich am Anfang des Programms eingerichtet, und ich habe auch nicht bis fast zum Ende des Programms gewartet, um `diskScanner` wieder zu schließen.

Stattdessen führe ich alle Aufgaben von `diskScanner` in schneller Folge nacheinander aus:

```
Scanner diskScanner =
    new Scanner(new File("GuestList.txt"));    //erste
llen
for (roomNum = 0; roomNum < 10; roomNum++) {
    guests[roomNum] = diskScanner.nextInt();    //lesen
}
diskScanner.close();                            //schli
eßen
```

Ich mache dasselbe mit den Objekten `keyboard` und `listOut`.

Ich veranstalte dieses Tänzchen mit Eingabe und Ausgabe, weil mein Programm `GuestList.txt` zweimal verwendet – einmal zum Auslesen von Zahlen und ein zweites Mal zum Schreiben von Zahlen. Wenn ich dabei nicht aufpasse, kann die zweimalige Verwendung von `GuestList.txt` zu einem Konflikt führen.

Stellen Sie sich dieses Programm vor:

```
// DIES IST SCHLECHTER CODE
import java.io.File;
import java.io.IOException;
import java.io.PrintStream;
import java.util.Scanner;
public class BadCode {

    public static void main(String args[])
                                throws IOException
    {
        int guests[] = new int[10];

        Scanner diskScanner =
            new Scanner(new File("GuestList.txt"));
        PrintStream listOut =
            new PrintStream("GuestList.txt");

        guests[0] = diskScanner.nextInt();
        listOut.print(5);

        diskScanner.close();
        listOut.close();
    }
}
```

Der Konstruktor `PrintStream` kommt, wie viele Methoden und Konstruktoren dieser Art, bei Dateien gleich zur Sache. Wenn die Datei `GuestList.txt` nicht gefunden wird, legt der Konstruktor eine neue an und ist bereit, dort hinein Werte zu schreiben. Wenn aber die entsprechende Datei bereits vorhanden ist, löscht der Konstruktor `PrintStream` sie und bereitet sich darauf vor, eine neue, leere Datei zu erstellen, in die er dann Werte schreiben kann. In der Klasse `BadCode` löscht der Aufruf des Konstruktors `new PrintStream` eine vorhandene Datei `GuestList.txt`. Dieses Löschen geschieht *vor* dem Aufruf von `diskScanner.nextInt()`. Damit hat `diskScanner.nextInt()` keine Chance mehr, irgendetwas in der ursprünglichen und nun nicht mehr vorhandenen Datei `GuestList.txt` zu lesen. Das ist gar nicht gut!

Um dieses Desaster zu vermeiden, trenne ich in [Listing 11.4](#) die beiden Verwendungen von `GuestList.txt` sorgfältig. Ich entwerfe zu Beginn des Listings `diskScanner`, dann lese ich die ursprüngliche Datei `GuestList.txt` aus und schließe `diskScanner`. Später, gegen Ende des Listings, baue ich `listOut` zusammen, schreibe dann in eine neue Datei `GuestList.txt` und schließe `listOut` wieder. Indem ich das Schreiben vollständig vom Lesen trenne, läuft alles sauber ab.



In [Listing 11.4](#) verweist die Variable `keyboard` nicht auf `GuestList.txt`, weshalb es zwischen `keyboard` und den anderen Variablen, die für die Eingabe oder Ausgabe zuständig sind, nicht zu Konflikten kommt. Alles geht gut, wenn Sie meiner normalen Routine folgen und `keyboard = new Scanner(System.in)` an den Anfang und `keyboard.close()` an das Ende des Programms setzen. Damit [Listing 11.4](#) aber einfach und einheitlich aufgebaut ist, habe ich den `keyboard`-Konstruktor und den Aufruf von `close` so nahe wie möglich an den Aufruf von `keyboard.nextInt` platziert.

Arrays aus Objekten

Im Java-Motel wird jetzt mit einer verbesserten Software für das Registrieren der Gäste gearbeitet. Diejenigen, die für den ersten Abschnitt dieses Kapitels zuständig waren, kratzen sich noch immer am Kopf und suchen nach dem besten Weg, um ihren Service zu verbessern. Nachdem sie nun immer mehr in den Bahnen der objektorientierten Programmierung denken, kommt ihnen eine Klasse `Room` (*Zimmer*) in den Sinn.

Ihre Frage nach dem Aussehen dieser Klasse lässt sich leicht beantworten. Die Instanz einer Klasse `Room` besitzt drei Eigenschaften: Anzahl der Gäste im Zimmer, Zimmerpreis und ein Raucher/Nichtraucher-Merkmal. [Abbildung 11.8](#) stellt die Situation grafisch dar.

Index	Gäste	Preis	Raucher
0	1	60.00	Nein
1	4	60.00	Nein
2	2	60.00	Ja
3	0	60.00	Ja
4	2	80.00	Nein
5	1	80.00	Ja
6	4	80.00	Ja
7	3	80.00	Ja
8	0	100.00	Nein
9	2	100.00	Ja

[Abbildung 11.8](#): Eine weitere abstrakte Darstellung der Zimmer im Java-Motel

[Listing 11.5](#) enthält den Code mit der Klasse `Room`. Wie versprochen, gibt es in jeder Instanz dieser Klasse drei Felder: `guests` (für die Anzahl der Gäste), `rate` (für den Zimmerpreis) und `smoking`. (Der Wert `false` weist im Boole'schen Feld `smoking` auf ein Nichtraucherzimmer hin.) Zusätzlich verfügt die gesamte Klasse `Room` über ein `static`-Feld mit dem Namen `currency` (*Währung*), das für die Formatierung der Zimmerpreise sorgt. (Welche Bedeutung das Wort `static` hat, können Sie in [Kapitel 10](#) nachlesen.)


```

import static java.lang.System.out;
import java.util.Scanner;
import java.text.NumberFormat;
public class Room {
    private int guests;
    private double rate;
    private boolean smoking;
    private static NumberFormat currency =
        NumberFormat.getCurrencyInstance();
    public void readRoom(Scanner diskScanner) {
        guests = diskScanner.nextInt();
        rate = diskScanner.nextDouble();
        smoking = diskScanner.nextBoolean();
    }
    public void writeRoom() {
        out.print(guests);
        out.print("t");
        out.print(currency.format(rate));
        out.print(" t");
        out.println(smoking ? "ja" : "nein");
    }
}

```

[Listing 11.5](#): So sieht ein Zimmer aus.

[Listing 11.5](#) enthält ein paar interessante Wendungen, die ich aber erst dann beschreibe, wenn Sie den gesamten Code in Aktion gesehen haben. Deshalb mache ich an dieser Stelle sofort mit dem Code weiter, der den Code in [Listing 11.5](#) aufruft. Nachdem Sie dann von Arrays gelesen haben, die aus Zimmern bestehen (und in [Listing 11.6](#) gezeigt werden), sollten Sie sich mit meiner Beschreibung von [Listing 11.5](#) beschäftigen.



Diese Warnung ist eine bewusste Wiederholung eines Gedankens aus den [Kapiteln 4](#), 7 und wer weiß wo sonst noch: Sie sollten sehr vorsichtig sein, wenn Sie die Typen `double` oder `float` verwenden, um Geldbeträge zu speichern. Berechnungen, die Sie mit `double` oder `float` durchführen, sind ungenau. Ausführliche Informationen zu diesem Thema finden Sie in [Kapitel 4](#).



Dieser Tipp hat überhaupt nichts mit Java zu tun. Wenn Sie zu denen gehören, die gerne ein Raucherzimmer hätten (mit dem `boolean`-Feld `smoking = true` in [Listing 11.5](#)), sollten Sie jemanden suchen, den Sie mögen – jemanden, der sich für drei Tage Urlaub nehmen kann. Diese Person muss dann 72 Stunden lang auf Sie aufpassen, während Sie auf das Rauchen verzichten. Während das Nikotin Ihren Körper verlässt, werden Sie zeitweilig ungenießbar sein, aber danach sollte alles in Ordnung sein. Und Ihr Freund wird sich wie ein wahrer Held fühlen.

Die Klasse »Room« verwenden

Sie benötigen nun ein Array aus Zimmern. Der Code, um so etwas zu erstellen, befindet sich in Listing 11.6. Er liest Daten aus der Datei `RoomList.txt` aus. ([Abbildung 11.9](#) gibt den Inhalt der Datei `RoomList.txt` wieder.)

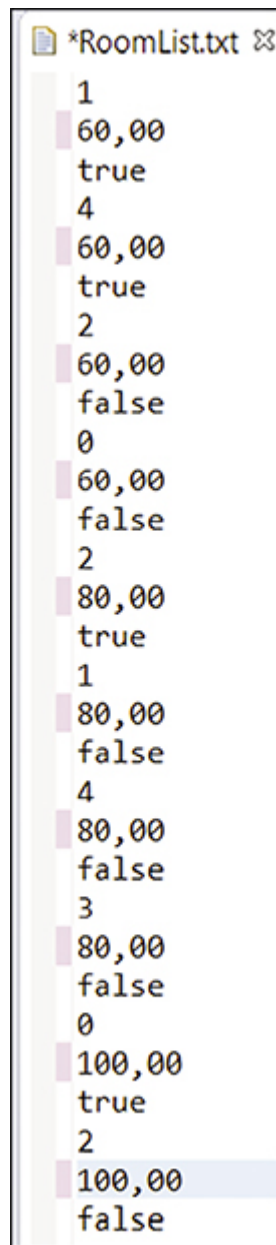
[Abbildung 11.10](#) zeigt, wie es aussieht, wenn der Code in [Listing 11.6](#) ausgeführt wird.

```
import static java.lang.System.out;
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
public class ShowRooms {
    public static void main(String args[])
                                throws IOException
    {
        Room rooms[];
        rooms = new Room[10];

        Scanner diskScanner =
            new Scanner(new File("RoomList.txt"));
        for (int roomNum = 0; roomNum < 10; roomNum++)
        {
            rooms[roomNum] = new Room();
            rooms[roomNum].readRoom(diskScanner);
        }
        out.println("ZimmertGästetPreist      Raucher?"
    );
};
```

```
        for (int roomNum = 0; roomNum < 10; roomNum++)
    ) {
        out.print(roomNum);
        out.print("t");
        rooms[roomNum].writeRoom();
    }
    diskScanner.close();
}
```

[Listing 11.6](#): Möchten Sie ein Zimmer besichtigen?



```
*RoomList.txt
1
60,00
true
4
60,00
true
2
60,00
false
0
60,00
false
2
80,00
true
1
80,00
false
4
80,00
false
3
80,00
false
0
100,00
true
2
100,00
false
```

[Abbildung 11.9](#): Eine Datei mit Daten von Zimmern

Zimmer	Gäste	Preis	Raucher?
0	1	60,00 €	ja
1	4	60,00 €	ja
2	2	60,00 €	nein
3	0	60,00 €	nein
4	2	80,00 €	ja
5	1	80,00 €	nein
6	4	80,00 €	nein
7	3	80,00 €	nein
8	0	100,00 €	ja
9	2	100,00 €	nein

[Abbildung 11.10](#): Der Code in [Listing 11.6](#) wird ausgeführt.

Sagen Sie über den Code in [Listing 11.6](#), was Sie wollen. So weit es mich betrifft, gibt es im gesamten Listing nur ein Element, um das Sie sich wirklich kümmern sollten. Im Gegensatz zu einem Array aus primitiven Werten erstellen Sie ein Array aus *Objekten*, in dem Sie drei Dinge erledigen müssen: die Array-Variable anlegen, das Array selbst erstellen und dann jedes einzelne Objekt im Array entwerfen. Dies ist anders als beim Erstellen eines Arrays aus `int`-Werten oder eines Arrays aus anderen primitiven Wertetypen. Wenn Sie ein Array aus primitiven Typen erstellen, kümmern Sie sich nur um die ersten beiden dieser drei Dinge.

Damit Sie den Sinn der folgenden Punkte besser verstehen, sollten Sie beim Lesen immer wieder einen Blick auf [Listing 11.6](#) und [Abbildung 11.11](#) werfen:

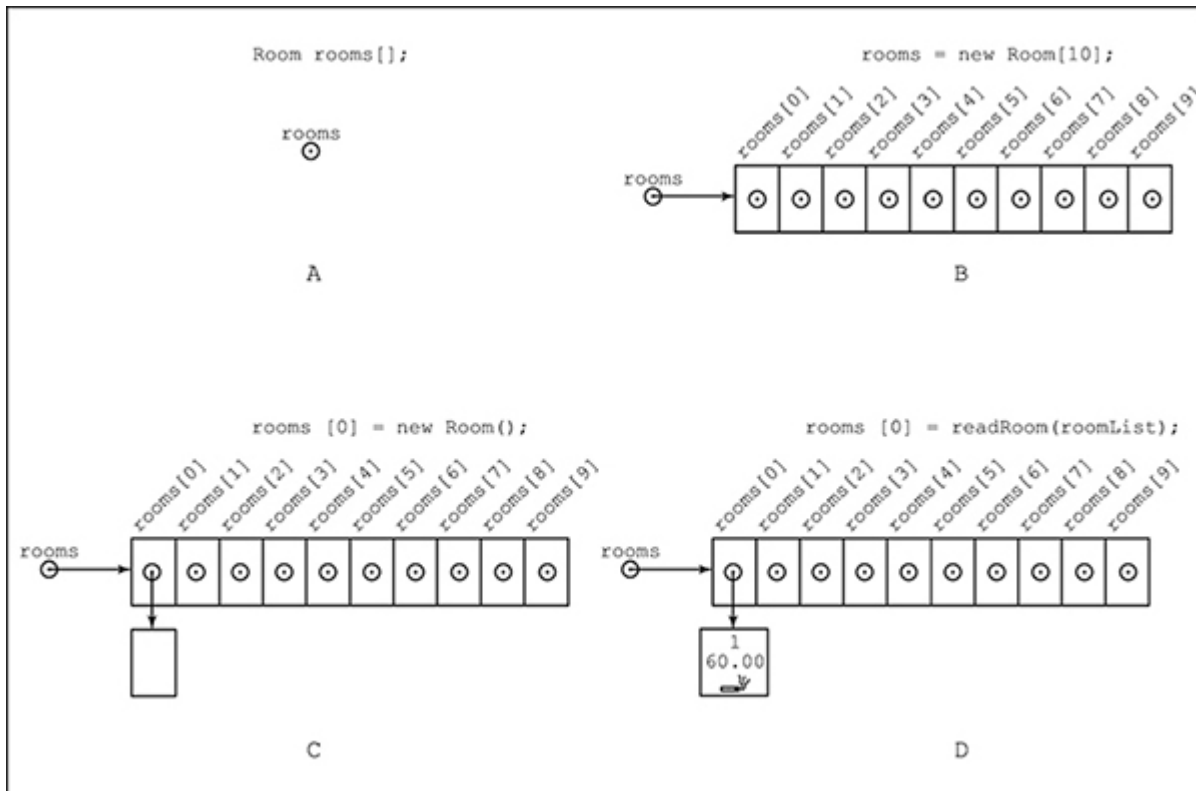


Abbildung 11.11: Die Schritte beim Erstellen eines Arrays aus Objekten

- ✓ **Room rooms[] ;:** Diese Deklaration erstellt eine Variable `rooms`. Diese Variable soll auf ein Array verweisen (was aber im Moment nicht der Fall ist, weil sie noch auf überhaupt nichts verweist).
- ✓ **rooms = new Room[10] ;:** Dieser Befehl reserviert im Arbeitsspeicher des Computers zehn Speicherplätze. Außerdem sorgt dieser Befehl dafür, dass die Variable `rooms` auf die Gruppe der Speicherplätze verweist. Jeder Speicherplatz soll später auf ein Objekt verweisen.
- ✓ **rooms[roomNum] = new Room() ;:** Diese Anweisung befindet sich in einer `for`-Schleife. Die Anweisung wird einmal für jede der zehn Zimmernummern ausgeführt. So lautet dieser Befehl beim ersten Durchlauf der Schleife `rooms[0] = new Room()` und sorgt dafür, dass der Speicherplatz `rooms[0]` auf ein tatsächliches Objekt (eine Instanz der Klasse `Room`) verweist.

Obwohl es technisch nicht zum Erstellen eines Arrays gehört, müssen Sie die Felder der einzelnen Objekte mit Werten füllen. Beim ersten Durchlaufen der Schleife lautet der Aufruf von `readRoom rooms[1].readRoom(diskScanner)`, was bedeutet: »Lies Daten aus der Datei `RoomList.txt` in die Felder (`guests`, `rate` und `smoking`) des Objekts `rooms[1]` ein.« Bei jedem Durchlaufen der Schleife erstellt das Programm ein neues Objekt und liest Daten in dessen Felder ein.

Sie können die Schritte so zusammenfassen, wie das beim Erstellen eines Arrays aus primitiven Werten möglich ist. Sie können zum Beispiel die ersten beiden Schritte in einem Rutsch ausführen lassen:

```
Room rooms[] = new Room[10];
```

Sie können aber auch einen Array-Initialisierer verwenden. (Eine Einführung in dieses Thema steht weiter vorn in diesem Kapitel im Abschnitt *Einen Array-Initialisierer verwenden*.)

Und noch ein Weg, um Zahlen schön zu gestalten

Es gibt viele Wege, um Zahlen »schön« aussehen zu lassen. Wenn Sie noch einmal einen Blick in frühere Kapitel werfen, sehen Sie zum Beispiel, dass [Listing 7.7](#) `printf` und [Listing 10.1](#) ein `DecimalFormat` verwenden. In [Listing 11.5](#) zeige ich einen Währungsbetrag an. Ich verwende die Klasse `NumberFormat` mit ihrer Methode `getCurrencyInstance`.

Wenn Sie die Formatierungsbefehle in den [Listings 10.1](#) und [11.5](#) miteinander vergleichen, werden Sie kaum Unterschiede feststellen:

✓ **Ein Listing verwendet einen Konstruktor; das andere Listing ruft `getCurrencyInstance` auf.** Die Methode `getCurrencyInstance` ist ein gutes Beispiel für das, was *Fabrikmethode* (englisch *Factory Method*) genannt wird. Bei einer Fabrikmethode handelt es sich um ein allgemein die Arbeit erleichterndes Werkzeug, mit dem Objekte für den

alltäglichen Gebrauch erstellt werden. Ständig wird Code benötigt, der Beträge in Euro anzeigt. Deshalb gibt es die Methode `getCurrencyInstance`, die einen Betrag im landesspezifischen Währungsformat und mit dem entsprechenden Währungssymbol (wie € in Deutschland oder \$ in den USA) versieht, ohne dass Sie gezwungen sind, `new DecimalFormat ("###0,00 €; (###0,00 €) ")` zu schreiben.

Eine Fabrikmethode gibt wie ein Konstruktor ein ganz neues Objekt zurück. Aber eine Fabrikmethode besitzt, anders als ein Konstruktor, keinen besonderen Status. Wenn Sie eine Fabrikmethode erstellen, können Sie ihr einen beliebigen Namen geben. Wenn Sie eine Fabrikmethode aufrufen, verwenden Sie das Schlüsselwort `new` nicht.

✓ **Ein Listing verwendet `DecimalFormat`; das andere Listing verwendet `NumberFormat`.** Bei einer Dezimalzahl handelt es sich um eine bestimmte Art von Zahl. (Es ist eine Zahl, die zu einem Zahlensystem gehört, das als Basis die 10 verwendet.) Dementsprechend ist die Klasse `DecimalFormat` eine Unterklasse der Klasse `NumberFormat`, wobei die `DecimalFormat`-Methoden spezialisierter sind, weshalb ich sie gerne einsetze. Sie sollten aber wissen, dass die Methode `getCurrencyInstance` der Klasse `DecimalFormat` schwieriger anzuwenden ist. Aus diesem Grund ziehe ich es vor, `NumberFormat` zu verwenden, wenn es um Programme geht, die mit Geld zu tun haben.

✓ **Beide Listings verwenden `format`-Methoden.** Letztendlich schreiben Sie so etwas wie `currency.format(rate)` oder `decFormat.format(average)`. Den Rest der Arbeit übernimmt dann Java.



Seit [Kapitel 4](#) streue ich immer wieder Warnungen gegen das Speichern von Währungswerten in Typen wie `double` und `float` ein. (Ich verweise hier ganz besonders auf [Kapitel 5](#).)

Wenn Sie genau mit Währungen rechnen müssen, verwenden Sie `int`, `long` oder am besten `BigDecimal`.

Der Bedingungsoperator

Listing 11.5 verwendet etwas Interessantes, das sich *Bedingungsoperator* nennt. Dieser Operator übernimmt drei Ausdrücke und gibt den Wert nur eines dieser Ausdrücke zurück. Der Bedingungsoperator ist wie eine `if`-Anweisung im Miniformat. Wenn Sie diesen Operator verwenden, sieht das so aus:

```
testBedingung ? ausdruck1 : ausdruck2
```

Der Computer wertet die Bedingung `testBedingung` aus. Wenn die Bedingung wahr ist, gibt der Computer den Wert von `ausdruck1` zurück, ansonsten wird der Wert von `ausdruck2` zurückgegeben.

Im Ausdruck

```
smoking ? "ja" : "nein"
```

prüft der Computer, ob `smoking` den Wert `true` hat. In diesem Fall steht der gesamte dreiteilige Ausdruck für den ersten String `"ja"`. Im anderen Fall steht der Ausdruck für den zweiten String `"nein"`.

In [Listing 11.5](#) sorgt der Ausdruck `out.println` dafür, dass entweder `ja` oder `nein` angezeigt werden. Welcher String schließlich auf dem Bildschirm erscheint, hängt davon ab, ob `smoking` den Wert `true` oder `false` hat.

Argumente auf der Befehlszeile

Vor langer, langer Zeit verwendeten Programmierer eine textbasierende Oberfläche. Um das `Displayer`-Beispiel in [Kapitel 3](#) auszuführen, wählten sie nicht im Menü einer wunderbar integrierten Entwicklungsumgebung `Run` aus. Stattdessen wurde in einem leer aussehenden Fenster ein Befehl eingetippt.

[Abbildung 11.12](#) gibt diese Vorgehensweise wieder. Ich habe in dieser Abbildung die Wörter `java Displayer` eingegeben, und

der Computer antwortet mit dem, was mein Programm ausgibt (den Wörtern `Sie werden Java lieben!`).

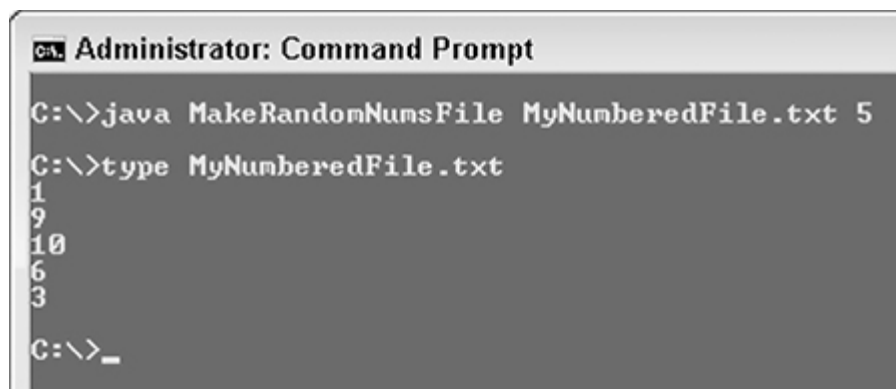


```
Administrator: Command Prompt
C:\>java Displayer
Sie werden Java lieben!
C:\>
```

[Abbildung 11.12](#): Wie langweilig!

Diese Art von Fenster hat abhängig von dem Betriebssystem, das Sie verwenden, viele Namen. In Windows wird ein Fenster dieser Art *Eingabeaufforderung* genannt. Beim Mac und in Linux heißt es *Terminal*, wobei einige Linux- und Unix-Versionen auch von einer *Shell* sprechen.

Also, in der »guten, alten Zeit« konnten Sie ein Programm schreiben, das zusätzliche Informationen benötigte, wenn Sie den Befehl für den Programmstart eingaben. [Abbildung 11.13](#) zeigt Ihnen, was ich damit meine.



```
Administrator: Command Prompt
C:\>java MakeRandomNumsFile MyNumberedFile.txt 5
C:\>type MyNumberedFile.txt
1
9
10
6
3
C:\>_
```

[Abbildung 11.13](#): Wenn Sie *MakeRandomNumsFile* starten, müssen Sie zusätzliche Informationen eingeben.

In [Abbildung 11.13](#) gibt der Programmierer `java MakeRandomNumsFile` ein, um das Programm `MakeRandomNumsFile` zu starten. Das reicht aber für einen Erfolg nicht aus, weshalb diesem ersten Teil noch weitere Informationen in Form von `MyNumberedFile.txt` und `5` folgen. Wenn das Programm `MakeRandomNumsFile` läuft, verwendet es diese zusätzlichen Informationen, um seinen Job zu erledigen. Beim nächsten Programmstart kann es sein, dass als zusätzliche Information nicht `MyNumberedFile.txt` und `5`, sondern `IrgendEtwas 28` oder `EinHaufenZahlen 2000` eingegeben werden muss. Es kommt vor, dass sich die Zusatzinformationen von Programmstart zu Programmstart ändern.

Deshalb lautet die nächste Frage: »Wie weiß ein Java-Programm, dass es bei jedem Programmstart zusätzliche Informationen verarbeiten muss?« Seitdem Sie angefangen haben, mit Java zu arbeiten, haben Sie es im Kopf der Methode `main` mit `String args[]` zu tun. Es ist nun höchste Zeit, dass Sie herausfinden, worum es dabei geht. Der Parameter `args[]` ist ein Array aus `String`-Werten. Diese `String`-Werte werden *Befehlszeilenargumente* genannt.

Befehlszeilenargumente in einem Java-Programm verwenden

Listing 11.7 zeigt, wie Befehlszeilenargumente im Code verwendet werden.

```
import java.util.Random;
import java.io.PrintStream;
import java.io.IOException;
public class MakeRandomNumsFile {

    public static void main(String args[])
                                throws IOException
    {
        Random generator = new Random();

        if (args.length < 2) {
            System.out.println
```

```

        ("Verwendung: MakeRandomNumsFile Dateiname
"
        + "Zahl");
        System.exit(1);
    }

    PrintStream printOut = new PrintStream(args[0])
;
    int numLines = Integer.parseInt(args[1]);

    for (int count = 1; count <= numLines; count++)
    {
        printOut.println(generator.nextInt(10) + 1);
    }
    printOut.close();
}
}

```

Listing 11.7: Eine Datei voller Zahlen erzeugen



Wenn ein Programm Befehlszeilenargumente erwartet, können Sie es nicht auf dieselbe Weise starten wie die meisten anderen Programme in diesem Buch. Wie Sie Befehlszeilenargumente an ein Programm übergeben, hängt von der IDE ab, die Sie verwenden. Sie finden auf der amerikanischen Website dieses Buches (www.allmycode.com/JavaForDummies) Anleitungen, um in verschiedenen IDEs Befehlszeilenargumente zu übergeben.

Wenn der Code in [Listing 11.7](#) ausgeführt wird, erhält das Array `args` Werte. Die Komponente `args[0]` des Arrays übernimmt automatisch den Wert `"MyNumberedFile.txt"`, und `args[1]` wird automatisch zu 5. Damit lautet der Zuweisungsbefehl des Programms wie folgt:

```

PrintStream printOut = new
    PrintStream("MyNumberedFile.txt");
int numLines = Integer.parseInt("5");

```

Das Programm erstellt eine Datei mit dem Namen `MyNumberedFile.txt` und setzt `num Lines` auf 5. Weiter hinten im Programm erzeugt das Programm fünf Zufallswerte und

schreibt diese Werte in die Datei `MyNumberedFile.txt`. Ich habe das Programm einmal ausgeführt und die Datei erhalten, die [Abbildung 11.14](#) zeigt.

```
6
3
10
10
8
```

[Abbildung 11.14](#): Eine Datei, die der Code in [Listing 11.14](#) erstellt hat



Wo können Sie, nachdem der Code in [Listing 11.7](#) abgelaufen ist, die Datei (`MyNumberedFile.txt`) auf Ihrer Festplatte finden? Ich möchte mich hier nicht auf eine Antwort festlegen. Wenn Sie eine IDE einsetzen, bei der die Programme als Projekte verwaltet werden, befindet sich die Datei irgendwo im Projektordner. Sie können aber [Listing 11.7](#) so ändern, dass es einen vollständigen Pfad zu einem Zielordner wie `"c:MyNumberFile.txt"` enthält.



In Windows enthalten die Namen von Dateipfaden umgekehrte Schrägstriche (die auch im deutschen Sprachgebrauch *Backslash* genannt werden). Wenn Sie in Java die Adresse eines Verzeichnisses auf einem Speichermedium angeben wollen, müssen Sie bei einem `String`-Literal einen doppelten und keinen einfachen Backslash für die Adressierung verwenden. Aus diesem Grund enthält `"c: MyNumberedFile.txt"` zwei dieser umgekehrten Schrägstriche. Im Gegensatz dazu enthalten Dateipfade in den Betriebssystemen Linux und Macintosh Schrägstriche. Um in einem Java-String einen Schrägstrich anzugeben, verwenden Sie einfach einen Schrägstrich. Um auf eine Datei im Dokumentenverzeichnis Ihres Macs zu verweisen, schreiben Sie

```
"/Users/IhrBenutzerName/Documents/MyNumberedFile.txt".
```

Achten Sie darauf, dass in [Listing 11.7](#) jedes Befehlszeilenargument ein `String`-Wert ist. Wenn Sie `args[1]` anschauen, sehen Sie die Zahl 5 – nein, Sie sehen den String "5", der eine Ziffer enthält. Unglücklicherweise können Sie diese "5" nicht zum Zählen verwenden. Um aus "5" einen `int`-Wert zu machen, müssen Sie die Methode `parseInt` anwenden (siehe noch einmal [Listing 11.7](#)).

Die Methode `parseInt` befindet sich in der Klasse `Integer`. Um die Methode `parseInt` aufzurufen, stellen Sie dem Methodennamen den Namen der Klasse `Integer` voran. Diese Klasse verfügt über viele praktische Methoden, um mit `int`-Werten zu arbeiten.



In Java ist `Integer` der Name einer Klasse und `int` der Name eines primitiven (einfachen) Typs. Diese beiden Objekte sind zwar miteinander verwandt, aber nicht dasselbe. Die Klasse `Integer` enthält Methoden und andere Werkzeuge, um mit `int`-Werten umzugehen.

Prüfen, ob die richtige Anzahl an Befehlszeilenparametern vorhanden ist

Was geschieht, wenn der Benutzer einen Fehler macht? Was, wenn der Benutzer in der ersten Zeile in [Abbildung 11.13](#) die Eingabe der Zahl 5 vergisst?

Dann weist der Computer `args[0]` zwar "MyNumberedFile.txt" zu, aber `args[1]` geht im wahrsten Sinne des Wortes leer aus – und das ist schlecht. Wenn der Computer dann den Befehl

```
int numLines = Integer.parseInt(args[1]);
```

erreicht, stürzt das Programm mit einer unfreundlichen `ArrayIndexOutOfBoundsException` ab.

Wie gehen Sie mit diesem Problem um? Sie können in [Listing 11.7](#) die Länge des Arrays überprüfen. Sie vergleichen `args.length` mit 2. Wenn das Array `args` weniger als zwei Komponenten enthält, zeigen Sie auf dem Bildschirm eine Meldung an und steigen mit dieser Bildschirmausgabe aus dem Programm aus:

```
Usage: MakeRandomNumsFile Dateiname Zahl
```

Diese Zeile sagt Ihnen, mit welchen Parametern Sie den Code in [Listing 11.7](#) starten müssen.



Der Code ist trotz der Überprüfung von `args.length` in [Listing 11.7](#) nicht sicher. Wenn der Benutzer fünf statt 5 eingibt, stürzt das Programm mit einer `NumberFormatException` ab. Das zweite Befehlszeilenargument darf kein Wort, sondern es muss eine Zahl (sogar eine Ganzzahl) sein. Ich könnte [Listing 11.7](#) um Befehle erweitern, die den Code narrensicher machen, aber Sie lernen in [Kapitel 12](#) einen viel besseren Weg kennen, um eine Prüfung von `NumberFormatException` vorzunehmen.



Wenn Sie mit Befehlszeilenargumenten arbeiten, können Sie auch einen `String`-Wert eingeben, der ein Leerzeichen enthält. Packen Sie den String einfach in doppelte Anführungszeichen. So können Sie zum Beispiel den Code in [Listing 11.7](#) auch mit den Argumenten `"My Big Fat File.txt" 7` ausführen.



Langsam geht die Sonne über den Arrays dieses Buches unter. Das nächste Kapitel handelt von etwas Neuem. Aber bevor Sie die Arrays verlassen, sollten Sie hierüber nachdenken: Ein Array ist eine Zeile aus Elementen, und nicht jedes Element passt in eine Zeile. Nehmen Sie die

ersten Beispiele dieses Kapitels, in denen es um das Motel geht. Die von 0 bis 9 durchnummerierten Zimmer des Motels sind nebeneinander (auf einer Zeile) aufgereiht. Was ist, wenn Sie expandieren? Sie kaufen sich ein Hotel mit 50 Stockwerken und 100 Zimmern je Stockwerk. Damit werden die Daten »viereckig«. Wir haben 50 Zeilen, und jede Zeile besteht aus 100 Elementen. Natürlich können Sie sich auch vorstellen, dass sich alle Zimmer in einer langen Zeile befinden, aber warum sollten Sie das tun? Was halten Sie davon, ein zweidimensionales Array zu haben? Dies ist ein »viereckiges« Array, in dem jede Komponente zwei Indexwerte besitzt – eine Zeilennummer und eine Spaltennummer. Leider sprengt es den Rahmen dieses Buches, zweidimensionale Arrays vorzustellen. Aber wenn Sie die amerikanische Website dieses Buches unter www.allmycode.com/JavaForDummies besuchen, können Sie einiges über diese Art von Arrays erfahren.

12

Sammlungen und Streams verwenden (wenn Arrays nicht mehr ausreichen)

In diesem Kapitel

- ▶ Sich den Begrenzungen von Arrays stellen
 - ▶ Mit vielen Objekten gleichzeitig umgehen
 - ▶ Die coolen Programmierfunktionen von Java verwenden
 - ▶ Code für Multicoreprozessoren entwickeln
-

[Kapitel 11](#) handelt von Arrays. Sie können mit einem Array viele Objekte gleichzeitig verwalten. Mit einem Programm zur Verwaltung eines Hotels sind Sie in der Lage, bei der Zimmerbelegung immer auf dem Laufenden zu sein. Sie können schnell herausfinden, mit wie vielen Personen ein Zimmer belegt ist, oder Sie können nach einem freien Zimmer suchen.

Arrays sind ganz schön, aber sie haben auch ernst zu nehmende Limitierungen. Stellen Sie sich vor, dass Sie die Namen von Kunden in einer vorher festgelegten Reihenfolge ablegen. Ihr Code enthält ein Array, das 100 Namen aufnehmen kann:

```
String name[] = new String[100];  
for (int i = 0; i < 100; i++) {  
    name[i] = new String();  
}
```

Alles ist gut – bis eines Tages Kunde Nummer 101 erscheint. Während Sie die Daten von Kunde 101 eingeben, hoffen Sie inständig, dass sich das Array mit 100 Komponenten erweitert, um den gewachsenen Anforderungen zu entsprechen.

Das geschieht nicht. Arrays vergrößern sich nicht. Ihr Programm stürzt mit einer `ArrayIndexOutOfBoundsException` ab.

Sie schwören sich, in Ihrem nächsten Leben nur noch Arrays mit einer Länge von 1.000 zu erstellen. Und als dieses »nächste Leben« anrollt, handeln Sie dementsprechend:

```
String name[] = new String[1000];  
for (int i = 0; i < 1000; i++) {  
    name[i] = new String();  
}
```

Aber in Ihrem nächsten Leben ändern sich die wirtschaftlichen Bedingungen und es kommt zu einer Rezession. Statt 101 Kunden haben Sie nur noch drei. Nun vergeuden Sie Platz für 1.000 Namen, wo Platz für drei Namen ausgereicht hätte.

Und wenn es nicht zu einer Rezession kommt? Dann verfügen Sie über ein Array mit einer Größe von 1.000. Sie nutzen davon vielleicht 825 Plätze. Die Komponenten mit den Indexen 0 bis 824 werden verwendet und die Komponenten mit den Indexen 825 bis 999 liegen brach und warten darauf, gefüllt zu werden.

Eines Tages taucht dann ein ganz neuer Kunde auf. Da Sie Ihre Kunden in einer ganz bestimmten Reihenfolge abgelegt haben (alphabetisch nach Nachnamen oder nach Postleitzahlen und so weiter), möchten Sie diesen Kunden in Ihrem Array an der richtigen Stelle unterbringen. Das Problem ist nun, dass Ihr Kunde ziemlich weit vorn im Array platziert werden muss – in der Komponente mit dem Index 7. Was geschieht nun?

Sie nehmen den Namen, der sich in der Komponente Nummer 824 befindet, und verschieben ihn in die Komponente 825. Sie nehmen den Namen, der sich in der Komponente Nummer 823 befindet, und verschieben ihn in die Komponente 824. Sie nehmen den Namen, der sich in der Komponente Nummer 822 befindet, und verschieben ihn in die Komponente 823. Dies setzen Sie fort, bis Sie den Namen in der Komponente Nummer 7 verschoben haben, und bringen jetzt endlich dort den Namen des neuen Kunden unter. So etwas tut richtig weh! Klar, dem Computer ist das ziemlich egal. Aber während Sie die ganzen Namen verschieben, verschwenden Sie Prozessorzeit, Strom und

alle möglichen Ressourcen – ganz zu schweigen von Ihrer Arbeitsleistung.

»In meinem nächsten Leben reserviere ich alle zwei Namen zwei leere Komponenten.« Und wenn sich dann Ihr Geschäft immer besser entwickelt, finden Sie heraus, dass drei vielleicht doch die bessere Wahl gewesen wäre.

Hilfe durch Sammelklassen

Die Themen der letzten paar Absätze sind nicht neu. Computerexperten arbeiten schon lange daran. Sie haben zwar bis heute noch keine Lösung gefunden, die auf alles passt, aber sie haben ein paar schlaue Tricks entdeckt.

In der Java-API gibt es eine Reihe von Klassen, die unter dem Begriff *Sammelklassen* bekannt geworden sind. Jede Sammelklasse enthält Methoden, um viele Werte zu speichern. Und jede Methode einer Sammelklasse verwendet Tricks. Zusammengefasst gilt für Sie, dass sich bestimmte Sammelklassen so effizient wie möglich um die Dinge kümmern, die in den letzten Absätzen zur Sprache gekommen sind. Wenn Sie beim Schreiben von Code auf Probleme obiger Art stoßen, können Sie Sammelklassen verwenden und die Methoden dieser Klassen aufrufen. Anstatt sich über einen Kunden zu ärgern, dessen Name an Position 7 gehört, rufen Sie einfach die Methode `add` einer Klasse auf. Diese Methode fügt den Namen an einer Position Ihrer Wahl ein und kümmert sich um alles, was an Störungen auftauchen könnte. Wenn alles gut verläuft, wird erfolgreich eingefügt, anderenfalls können Sie zumindest sicher sein, dass der Code versucht, sein Bestes zu geben.

Eine »ArrayList« verwenden

Eine der vielseitigsten Sammelklassen Javas ist die `ArrayList`. [Listing 12.1](#) zeigt Ihnen, wie sie arbeitet.

```
import static java.lang.System.out;
import java.util.Scanner;
import java.io.File;
```

```

import java.io.IOException;
import java.util.ArrayList;
public class ShowNames {
    public static void main(String args[])
        throws IOException {
        ArrayList<String> people =
            new ArrayList<String>();
        Scanner diskScanner =
            new Scanner(new File("names.txt"));
        while (diskScanner.hasNext()) {
            people.add(diskScanner.nextLine());
        }
        people.remove(0);
        people.add(2, "Jim Newton");

        for (String name : people) {
            out.println(name);
        }
        diskScanner.close();
    }
}

```

[Listing 12.1](#): Mit einer Java-Sammlung arbeiten

[Abbildung 12.1](#) zeigt die Beispieldatei `name.txt`. Der Code in [Listing 12.1](#) liest diese Datei aus und gibt die Inhalte so wieder, wie es [Abbildung 12.2](#) zeigt.

Die interessanten Dinge geschehen, wenn Sie die Methoden `add` und `remove` ausführen. Die Variable `people` verweist auf das Objekt `ArrayList`. Wenn Sie die Methode `remove` dieses Objekts über

```
people.remove(0);
```



Abbildung 12.1: Mehrere Namen in einer Datei



Abbildung 12.2: Der Code in Listing 12.1 ändert einige Namen.

aufrufen, entfernen Sie einen Eintrag aus der Liste. In diesem Fall löschen Sie jeden Wert an der Anfangsposition der Liste (der Position mit der Nummer 0). In Listing 12.1 wird auf diese Weise Barry Burd aus der Liste entfernt.

Damit bleiben nur noch acht Namen in der Liste zurück, aber dann sorgt der Befehl

```
people.add(2, "Jim Newton");
```

dafür, dass ein Name an Position 2 eingefügt wird. (Nachdem Barry Burd gelöscht worden ist, befindet sich Harry Spoonswagler an Position 2. Nun wird

Harry Spoonswagler an Position 3 verschoben und Jim Newton zur Nummer 2 gemacht.)

Beachten Sie, dass ein `ArrayList`-Objekt über zwei verschiedene `add`-Methoden verfügt. Die Methode, die Jim Newton hinzufügt, weist zwei Parameter auf: eine Positionsnummer und einen Wert, der hinzugefügt wird. Die andere `add`-Methode

```
people.add(diskScanner.nextLine());
```

enthält nur einen Parameter. Dieser Befehl übernimmt den Namen, den er auf einer Zeile der Eingabedatei findet, und fügt diesen Namen dem Ende der Liste hinzu. (Die Methode `add` mit nur einem Parameter fügt ihren Wert immer dem Ende des aktuellen `ArrayList`-Objekts hinzu.)

Die letzten Zeilen von [Listing 12.1](#) enthalten eine erweiterte `for`-Schleife. Diese Schleife hat, wie die Schleife in [Listing 11.3](#), die Form

```
for (Variablentyp variablenName : Wertebereich)
```

In [Listing 12.1](#) ist der Variablentyp `String`, der Variablenname lautet `name` und der Wertebereich umfasst die Objekte, die in der Sammlung `people` gespeichert sind. (Wenn also die Sammlung `people` neun Werte enthält, durchläuft die `for`-Schleife neun Iterationen.) Der Befehl in der Schleife zeigt bei jedem Durchlauf auf dem Bildschirm einen Namen an.

Generische Typen verwenden

Schauen Sie sich noch einmal [Listing 12.1](#) an und achten Sie auf die ungewöhnliche Deklaration der `ArrayList`:

```
ArrayList<String> people = new ArrayList<String>();
```

Seit Java 5.0 ist jede Sammelklasse *generifiziert*. Dieses fürchterliche Wort bedeutet, dass jede Deklaration einer Sammlung Elemente in spitzen Klammern wie `<String>` enthalten sollte. Die Komponente, die von `<` und `>` eingeschlossen wird, sagt Java, welche Art von Werten die neue Sammlung enthält. So informieren zum Beispiel in [Listing 12.1](#) die Wörter `ArrayList <String> people` Java darüber, dass es sich bei

`people` um viele Strings handelt. Dies bedeutet, dass `people` `String`-Objekte enthält.



Die meisten Informationen dieses Abschnitts können nur mit Java-Versionen ab 5.0 in die Praxis umgesetzt werden. Sie können generische Typen (seltener auch *Generics* genannt) nicht in Java-Versionen vor 5.0 verwenden. Der Kasten *Alles über generische Typen* enthält (in diesem Kapitel) weitergehende Informationen zu diesem Thema, und in [Kapitel 2](#) können Sie mehr über Javas Versionsnummerierung nachlesen.

In [Listing 12.1](#) besagen die Wörter `ArrayList<String>` `people`, dass die Variable `people` nur auf eine Sammlung von `String`-Werten verweisen kann. Also wird von diesem Zeitpunkt an jeder Verweis der Sammlung `people` auf ein Element nur noch als `String` behandelt. Wenn Sie

```
people.add(new Room());
```

schreiben, verschluckt sich der Compiler an Ihrem Code und spuckt ihn aus, weil ein `Room` (erstellt in [Kapitel 11](#)) nicht dasselbe ist wie ein `String`. (Zu solch einer Aktion des Compilers kommt es selbst dann, wenn er Zugriff auf den Code der Klasse `Room` hat.) Aber der Befehl

```
people.add("George Gow");
```

geht in Ordnung. Da "George Gow" vom Typ `String` ist, ist der Compiler glücklich.



Java 7 und spätere Versionen kennen eine Funktion, die es Ihnen erlaubt, generische Deklarationen abzukürzen. Sie können in [Listing 12.1](#) auch `ArrayList<String>` `people = new ArrayList<>()` schreiben, ohne das Wort `String` in der Deklaration zu wiederholen. Das Symbol `<>` ohne Inhalt in den Klammern wird auch *Diamantoperator* genannt. Der Diamantoperator erspart es Ihnen, immer wieder `<String>` schreiben zu müssen.

Alles über generische Typen

Eines der Ziele beim Entwerfen von Java war es, die Sprache so einfach wie möglich zu halten. Bei der Entwicklung von Java wurden einige der unnötig komplizierten Funktionen von C++ genommen und auf den Müll geworfen. Das Ergebnis war eine elegante und geschmeidige Sprache. Es hieß sogar, dass diese Sprache zu geschmeidig sei. Nach einigen Jahren der Diskussion wurde Java ein wenig komplizierter. Bis 2004 hatte Java `enum`-Typen, erweiterte `for`-Schleifen, statischen `Import` und einige andere interessante Funktionen erhalten. Aber am meisten wurde über die Einführung von generischen Typen gesprochen.

```
ArrayList<String> people = new ArrayList<String>();
```

Die Verwendung von etwas wie `<String>` war in Java 5.0 neu. Im Java davor haben Sie Folgendes geschrieben:

```
ArrayList people = new ArrayList();
```

Damals war eine `ArrayList` in der Lage, so gut wie alles zu speichern, was Sie dort hineinpackten – eine Zahl, ein `Konto`, einen `Room`, einen `String`, einfach alles. Die Klasse `ArrayList` war sehr vielseitig, aber diese Vielseitigkeit führte auch zu Kopfschmerzen. Sie konnten zwar alles in einer `ArrayList` unterbringen, aber es war schwierig vorherzusagen, was Sie von ihr erhalten. So war es nicht möglich, einfach Code zu schreiben, der davon ausging, dass bestimmte Arten von Werten in einer `ArrayList` gespeichert waren. Hier ein Beispiel:

```
ArrayList things = new ArrayList();
things.add(new Konto());
Konto meinKonto = things.get(0);
//DIESEN CODE NICHT VERWENDEN. ER IST SCHLECHT.
```

In der dritten Zeile nimmt sich der Aufruf von `get(0)` den ersten Wert in der Sammlung `things`. Der Aufruf von `get(0)` geht in Ordnung, aber dann verschluckt sich der Compiler an der versuchten Zuweisung zu `meinKonto`. Diese dritte Zeile ruft eine Meldung hervor, die besagt, dass nichts von dem, was Sie aus der Liste `things` erhalten, in der Variablen `meinKonto` abgelegt werden kann. Zu dieser Meldung kommt es, weil der Compiler, wenn er diese dritte Zeile erreicht, vergessen hat, dass das Element, das auf der zweiten Zeile hinzugefügt wird, vom Typ `Konto` ist!

Dieses Problem wurde durch die Einführung der generischen Typen behoben:

```
ArrayList<Konto> things = new ArrayList<Konto>();
things.add(new Konto());
Konto meinKonto = things.get(0);
//VERWENDEN SIE DIESEN CODE. ER GEHT IN ORDNUNG:
```


Dadurch, dass Sie `<Konto>` an zwei Stellen hinzufügen, wird der Compiler darüber informiert, dass `things` nur Instanzen von `Konto` speichert. Deshalb erhalten Sie beim letzten Code in der dritten Zeile von der Sammlung `things` einen Wert. Da `things` nur `Konto`-Objekte enthält, können Sie dafür sorgen, dass `meinKonto` auf diesen neuen Wert verweist.

Seit Java 5.0 gibt es in Java generische Typen. Aber schon kurz nach dem Erscheinen von Java 5.0 fiel Programmierern auf, wie umständlich der Code für generische Typen sein kann. Schließlich sind Sie in der Lage, generische Typen innerhalb von generischen Typen zu erstellen. Eine `ArrayList` kann viele Arrays enthalten, von denen jedes eine `ArrayList` sein kann. So können Sie schreiben:

```
ArrayList<ArrayList<String>[]> mess =  
    new ArrayList<ArrayList<String>[]>  
    ();
```

Die Wiederholungen in der Deklaration von `mess` bereiten mir Kopfschmerzen! Um Unschönheiten dieser Art zu vermeiden, enthalten Java 7 und spätere Versionen den *Diamantoperator* `<>`. Dieser Operator weist Java an, das wiederzuverwenden, was Sie im davor stehenden Teil der Deklaration des generischen Typs untergebracht haben. Im folgenden Beispiel sagt `<>` Java, dass `<ArrayList<String>[]>` wiederverwendet werden soll, obwohl Sie `<ArrayList<String>[]>` nur einmal geschrieben haben. Und so sieht der abgespeckte Java-Code dann aus:

```
ArrayList<ArrayList<String>[]> mess =  
    new ArrayList<>();
```

In Java 7 oder später können Sie beide Versionen der `mess`-Deklaration schreiben – die alte, scheußliche Deklaration, in der `ArrayList<String>[]` zweimal vorkommt, oder die modernisierte (weniger schreckliche) mit dem Diamantoperator und dem nur einmal vorkommenden `ArrayList<String>[]`.

Auch der modernisierte Code ist noch recht kompliziert, aber ohne die ganzen Wiederholungen von `ArrayList<String>[]` ist er weniger umständlich. Der Diamantoperator von Java 7 nimmt Ihnen gleichzeitig die Möglichkeit, etwas Falsches zu kopieren und damit im Code einen großen Fehler zu erzeugen.

Testen, ob noch mehr Daten vorhanden sind

Hier kommt eine erfreuliche Überraschung. Bei einem Programm wie dem, das weiter vorn in [Listing 12.1](#) gezeigt wird, müssen Sie

nicht wissen, wie viele Namen es in der Eingabedatei gibt. Wenn Sie wüssten, wie viele Namen in der Datei stehen, könnten Sie vielleicht auf die Idee kommen, auf die leicht zu erweiternde Klasse `ArrayList` zu verzichten. Wenn Sie das machen, durchlaufen Sie so lange Schleifen, bis die vorgegebene Anzahl an Namen (in unserem Beispiel genau neun) gelesen worden sind, während Sie anderenfalls so lange Schleifen drehen können, bis Ihnen die Daten ausgehen.

Die Klasse `Scanner` kennt verschiedene Methoden wie `hasNextInt`, `hasNextDouble` und das gute alte `hasNext`. Jede dieser Methoden prüft nach, ob noch mehr Eingabedaten vorhanden sind. In diesem Fall gibt die Methode `true` zurück. Anderenfalls wird `false` zurückgegeben.

[Listing 12.1](#) verwendet die überall einsetzbare Methode `hasNext`. Diese Methode gibt `true` zurück, solange es bei den Eingabedaten des Programms noch etwas zu lesen gibt. Nachdem das Programm in der letzten Zeile von [Abbildung 12.1](#) Hugh R. DaReader aufgenommen hat, gibt der nächste Aufruf von `hasNext` `false` zurück. Dies beendet die `while`-Schleife und der Computer stürzt sich auf den Rest des Codes in [Listing 12.1](#).

Die Methode `hasNext` ist sehr praktisch. Sie ist sogar so praktisch, dass sie Teil eines größeren Konzepts ist, das unter dem Namen *Iterator* bekannt geworden ist. Iteratoren sind feste Bestandteile aller Java-Sammelklassen.

Einen Iterator verwenden

Ein Iterator gibt nacheinander die Werte einer Sammlung aus. Um aus einer Sammlung einen Wert zu erhalten, rufen Sie die Methode `next` des Iterators auf. Um herauszufinden, ob es in der Sammlung noch weitere Werte gibt, rufen Sie die Methode `hasNext` des Iterators auf. [Listing 12.2](#) verwendet einen Iterator, um die Namen von Personen anzuzeigen.

```
import static java.lang.System.out;
import java.util.Iterator;
```

```

import java.util.Scanner;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
public class ShowNames {
    public static void main(String args[])
        throws IOException {
        ArrayList<String> people =
            new ArrayList<String>();
        Scanner diskScanner =
            new Scanner(new File("names.txt"));
        while (diskScanner.hasNext()) {
            people.add(diskScanner.nextLine());
        }
        people.remove(0);
        people.add(2, "Jim Newton");

        Iterator<String> iterator = people.iterator()
;
        while (iterator.hasNext()) {
            out.println(iterator.next());
        }

        diskScanner.close();
    }
}

```

[Listing 12.2](#): Durch eine Sammlung iterieren

Sie können am Ende von [Listing 12.1](#) die erweiterte `for`-Schleife durch den fett gedruckten Code in [Listing 12.2](#) ersetzen. Wenn Sie dies machen, erhalten Sie dieselbe Ausgabe wie zuvor (siehe [Abbildung 12.2](#)). In [Listing 12.2](#) erstellt die erste fett gedruckte Zeile aus der Sammlung `people` einen Iterator. Die zweite und die dritte Zeile rufen die Methoden `hasNext` und `next` des Iterators auf, um sich alle Objekte der Sammlung `people` zu holen – bei jedem Durchgang der Schleife eines. Diese Zeilen zeigen alle Werte der Sammlung `people` an.

Welches ist nun die bessere Lösung? Eine erweiterte `for`-Schleife oder ein Iterator? Java-Programmierer bevorzugen die erweiterte `for`-Schleife, weil sie weniger Gepäck mit sich herumschleppt – es gibt kein `iterator`-Objekt, das von einer

Codezeile zur nächsten weitergereicht werden muss. Aber wie Sie weiter hinten in diesem Kapitel sehen, lassen sich die meisten erweiterten Programmierfunktionen auch später noch aktualisieren, modernisieren, genauer einstellen und neu aufbauen. Auf dem Weg, Ihren Code zu verbessern, ist noch kein Ende in Sicht.

Die vielen Sammelklassen Javas

Die Klasse `ArrayList`, die ich in vielen Beispielen dieses Kapitels verwende, ist nur die Spitze des Eisbergs der Java-Sammlungen. Die Java-Bibliothek enthält viele Sammelklassen, die alle ihre Vorteile haben. [Tabelle 12.1](#) enthält ein paar Beispiele.

Jede Sammelklasse hat einen eigenen Satz an Methoden, die die Methoden ergänzen, die jede Sammelklasse von `AbstractCollection`, der Urahnin aller Sammelklassen, erbt.



Wenn Sie herausfinden wollen, welche Sammelklasse Ihren Ansprüchen am besten entspricht, besuchen Sie die Seiten mit den Dokumentationen zu diesem Thema (docs.oracle.com/javase/8/docs/api).

Klassenname	Merkmale
<code>ArrayList</code>	Ein größenveränderbares Array
<code>LinkedList</code>	Eine Liste mit Werten, die alle ein Feld haben, das auf den nächsten Wert in der Liste zeigt
<code>Stack</code>	Eine Struktur, die von unten nach oben wächst. Die Struktur ist für den Zugriff auf den obersten Wert optimiert. Sie können oben leicht Werte hinzufügen oder von dort entfernen.
<code>Queue</code>	Eine Struktur, die an einem Ende wächst. Die Struktur ist dafür optimiert worden, an einem Ende Werte hinzuzufügen und am anderen Ende Werte zu entfernen.

Klassenname	Merkmale
PriorityQueue	Eine Struktur wie eine Warteschlange, die es zulässt, dass Komponenten mit einer höheren Priorität nach vorne verschoben werden
HashSet	Eine Sammlung, die keine doppelten Werte enthält
HashMap	Eine Sammlung von Schlüssel/Wert-Paaren

Tabelle 12.1: Ein paar Sammelklassen

Neu in Java 8: funktionale Programmierung

In der Zeit von 1953 bis 1957 entwickelten John Backus und andere die Programmiersprache *FORTRAN*, die das Basisgerüst für die Programmiersprachen des 20. Jahrhunderts gelegt hat. Dieses Gerüst wird wegen seiner »Mache dies und dann das«-Natur auch *imperative Programmierung* genannt.

Ein paar Jahre nach dem Aufstieg von FORTRAN entwickelte John McCarthy eine Programmiersprache mit dem Namen *Lisp*. Das Lisp zugrunde liegende Gerüst ist, anders als bei FORTRAN, eine funktionale Programmierung. Sie vermeiden in einer rein funktionalen Programmierung das Schreiben von »Tue dies und dann das«. Stattdessen schreiben Sie Dinge wie »Hier steht, wie du das hier in dieses dort umwandelst, wenn du so weit bist, etwas umzuwandeln.«

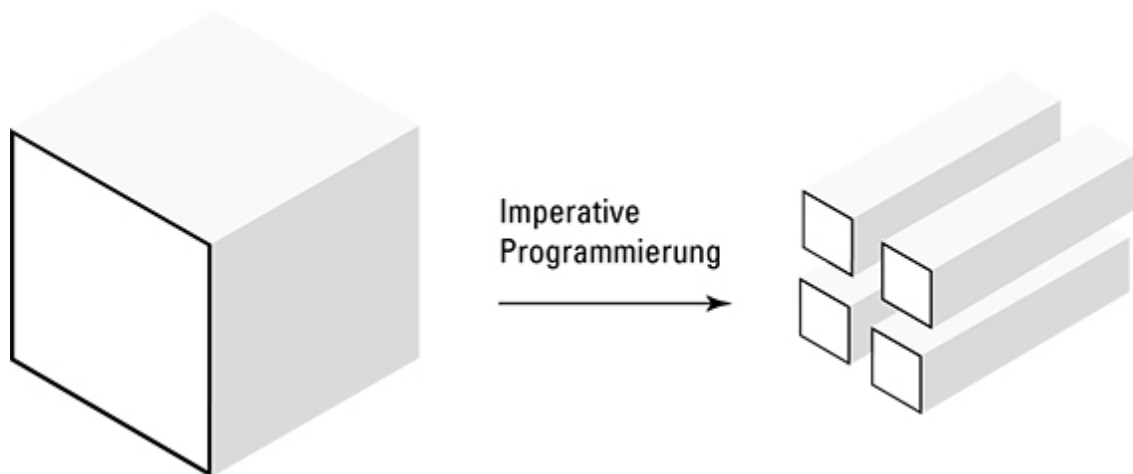
Aus irgendwelchen Gründen hat sich letztendlich die imperative Programmierung durchgesetzt. Deshalb ist auch Java grundsätzlich eine imperative Programmiersprache. Aber neuerdings erlebt die funktionale Programmierung eine Art Wiedergeburt als leistungsfähige und sinnvolle Möglichkeit, um über Code nachzudenken.

Ich möchte Sie jetzt nicht damit langweilen, dass ich die funktionale Programmierung im Einzelnen beschreibe, und

beginne diesen Abschnitt lieber mit einer Analogie. Danach zeige ich Ihnen im Rest dieses Kapitels einige Beispiele mit Java 8.

Die Analogie, die ich verwende, um funktionale Programmierung zu beschreiben, ist ziemlich grob gestrickt. Einer meiner Freunde hat sie als »weit hergeholt« bezeichnet. Er erinnerte mich daran, dass die Analogie auf viele verschiedene Programmierungsumgebungen und nicht nur auf die funktionale Programmierung zuträfe. Trotzdem bin ich der Meinung, dass die folgende Analogie hilfreich ist:

Stellen Sie sich ein Programmierproblem als Würfel vor und stellen Sie sich weiter vor, dass die imperative Programmierungslösung eine Möglichkeit ist, den Würfel in überschaubare Teile zu zerlegen (siehe [Abbildung 12.3](#)).



[Abbildung 12.3](#): Imperative Programmierung zerlegt Probleme.

Bis 2007 war alles in Ordnung. Dann wurden zum ersten Mal Computer mit Multicoreprozessoren an Endbenutzer verkauft. Ein Multicoreprozessor kann mehr als einen Befehl gleichzeitig ausführen. [Abbildung 12.4](#) zeigt, was geschieht, wenn Sie versuchen, ein imperatives Programm in einen Multicoreprozessor zu zwingen.

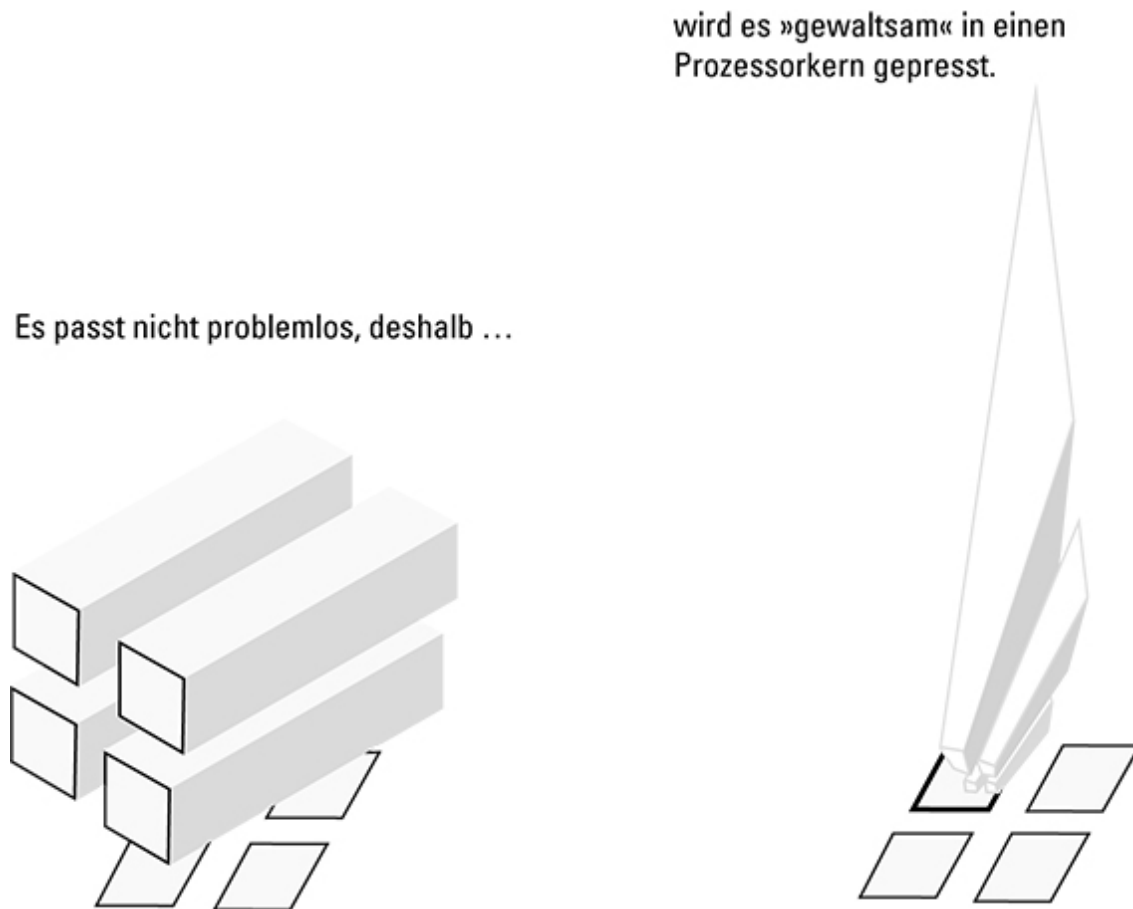


Abbildung 12.4: Die Teile eines imperativen Programms passen nicht nahtlos in einen Multicorechip.

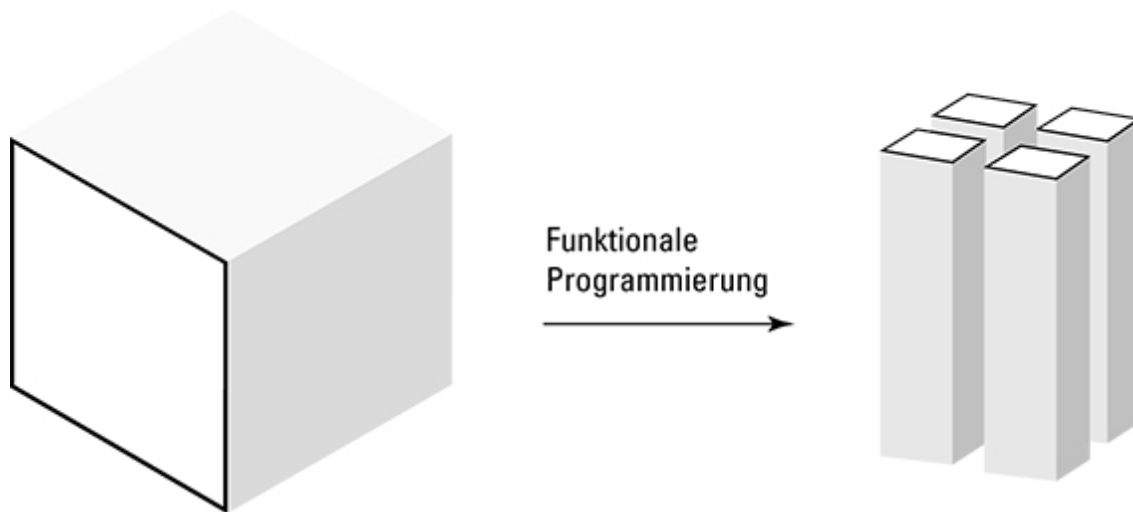
Um aus einem Prozessor mit vier Kernen das Optimum herauszuholen, teilen Sie Ihren Code in vier Bereiche auf – einen für jeden Kern. Aber bei der imperativen Programmierung passen die einzelnen Teile nicht problemlos in die Kerne des Prozessors.



Bei der imperativen Programmierung tauschen sich die Teile Ihres Codes untereinander aus. So könnten alle Teile den Preis der Oracle-Aktie (Ticker-Symbol ORCL) aktualisieren. Diese gleichzeitigen Aktualisierungen verheddern sich. Das ist so, als wenn mehrere Jungs unabhängig voneinander dasselbe Mädchen fragen, ob es mit ihnen auf den Abi-Ball ginge. So etwas kann nicht gut gehen. Sie machen eine ähnliche Erfahrung, wenn Sie auf die Schaltfläche KAUFEN einer Website klicken, um dann zu erfahren, dass die Ware,

die Sie erwerben wollten, nicht mehr vorrätig ist. Irgendjemand hat einen Einkauf abgeschlossen, während Sie noch die Daten Ihrer Kreditkarte eingegeben haben. Zu viele Kunden waren gleichzeitig hinter demselben Artikel her.

[Abbildung 12.3](#) schlägt vor, dass Sie Ihren Code bei der imperativen Programmierung aufteilen. Auch die funktionale Programmierung zerlegt den Code in einzelne Teile, aber dies geschieht hier entlang bestimmter Strukturen (siehe [Abbildung 12.5](#)). Die gute Nachricht dabei ist, dass diese Teile problemlos in die Kerne des Prozessors passen (siehe [Abbildung 12.6](#)).



[Abbildung 12.5](#): Das Problem entlang bestimmter Linien zerteilen.

Jedes Teil passt genau in einen Kern.

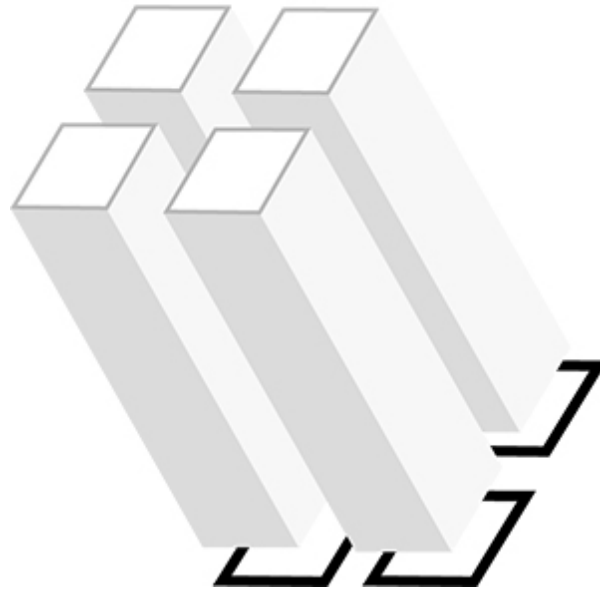


Abbildung 12.6: Die Teile passen bei einer funktionalen Programmierung genau in einen Multicorechip.

Ein Problem auf die altbewährte Weise lösen

In [Kapitel 11](#) verwenden Sie Arrays, um das Java-Motel zu verwalten. Dieses Unternehmen mussten Sie abhaken. Sie haben sich aus dem Hotelgeschäft zurückgezogen. (Sie erzählen zwar überall, dass Sie sich weiterentwickeln wollen, aber in Wirklichkeit hat das Hotel nur viel Geld gekostet. Deshalb sind Sie mit dem Ding in die Insolvenz geschliddert.)

Seitdem Sie das Hotelgewerbe aufgegeben haben, beschäftigen Sie sich mit Onlineverkäufen. Heute unterhalten Sie eine Website, auf der Bücher, DVDs und andere mit Inhalten dieser Art zusammenhängende Objekte vertrieben werden.

In Ihrer Welt sieht der Verkauf (englisch *Sale*) eines einzelnen Objekts so ähnlich aus wie das, was [Listing 12.3](#) enthält. Zu jedem Verkauf gehören ein Objekt und ein Preis.

```
public class Sale {  
    String item;
```

```

        double price;

        public Sale(String item, double price) {
            this.item = item;
            this.price = price;
        }
    }
}

```

[Listing 12.3](#): Die Klasse *Sale*

Um die Klasse `Sale` zu verwenden, erstellen Sie ein kleines Programm. Dieses Programm summiert die Beträge auf, die über die DVD-Verkäufe hereingekommen sind. [Listing 12.4](#) gibt das Programm wieder.

```

import java.text.NumberFormat;
import java.util.ArrayList;
public class TallySales {
    public static void main(String[] args) {
        ArrayList<Sale> sales = new ArrayList<Sale>
();
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();
        fillTheList(sales);
        double total = 0;
        for (Sale sale : sales) {
            if (sale.item.equals("DVD")) {
                total += sale.price;
            }
        }
        System.out.println(currency.format(total));
    }
    static void fillTheList(ArrayList<Sale> sales) {
        sales.add(new Sale("DVD", 15.00));
        sales.add(new Sale("Book", 12.00));
        sales.add(new Sale("DVD", 21.00));
        sales.add(new Sale("CD", 5.25));
    }
}

```

[Listing 12.4](#): Die Klasse *Sale* verwenden

In [Kapitel 11](#) durchlaufen Sie mit einem erweiterten `for`-Befehl ein Array. [Listing 12.4](#) hat einen eigenen erweiterten `for`-Befehl, der aber in diesem Listing die Werte einer Sammlung durchläuft. Bei

jedem dieser Werte handelt es sich um einen Verkauf. Die Schleife überprüft immer wieder die einzelnen Verkäufe, um herauszufinden, ob es sich bei dem verkauften Objekt um eine DVD handelt. In diesem Fall fügt der Code den Verkaufspreis der Gesamtsumme hinzu. Ausgegeben werden 36,00 € – die Summe wird im Währungsformat angezeigt.

Das Szenario in [Listing 12.4](#) ist nichts Besonderes. Sie verfügen über eine Sammlung von Dingen (wie Verkäufen). Sie gehen schrittweise durch die Dinge in der Sammlung und suchen das Ding, das bestimmten Kriterien entspricht (zum Beispiel Bestandteil des Verkaufs einer DVD zu sein). Sie greifen sich einen bestimmten Wert (wie den Verkaufspreis), der zu dem Ding gehört. Dann machen Sie mit den Werten, die Sie übernommen haben, etwas Sinnvolles (beispielsweise addieren Sie sie).

Hier weitere Beispiele:

- ✓ Schauen Sie sich die Liste mit Ihren Mitarbeitern an und zahlen Sie jedem mit einem Leistungsindex von 3 oder besser 100 Euro Prämie.
- ✓ Durchforsten Sie Ihre Kundenliste. Senden Sie jedem Kunden, der Interesse am Kauf eines Smartphones gezeigt hat, eine E-Mail mit den Sonderangeboten des Monats.
- ✓ Durchlaufen Sie die Liste mit den bisher entdeckten Planeten. Finden Sie für jeden Planeten der M-Klasse die Wahrscheinlichkeit heraus, ob es dort intelligentes Leben gibt. Berechnen Sie dann den Durchschnitt der Wahrscheinlichkeiten.

Zu diesen Szenarien kommt es so häufig, dass es sich lohnt, immer bessere Lösungswege zu finden. Einer davon ist der Einsatz der in Java 8 neuen funktionalen Programmierung.

Streams

Der Abschnitt *Einen Iterator verwenden* stellt Iteratoren vor. Sie verwenden die Methode `next` eines Iterators, um die Werte einer Sammlung auszugeben. Java 8 führt dieses Konzept mit der Vorstellung eines Streams einen Schritt weiter. Ein *Stream*

(deutsch *Strom*) ist wie ein Iterator, bei dem Sie keine Methode `next` aufrufen müssen. Nachdem ein Stream erstellt worden ist, spuckt er automatisch die Werte einer Sammlung aus.

Wie funktioniert so etwas in einem Java-Programm? Wie erstellen Sie einen Stream, der Werte ausgibt? Woher weiß der Stream, bei welchem Wert er anfangen und wohin er die Werte ausgeben soll? Antworten auf diese und andere Fragen gibt es in den nächsten Abschnitten.

Lambda-Ausdrücke

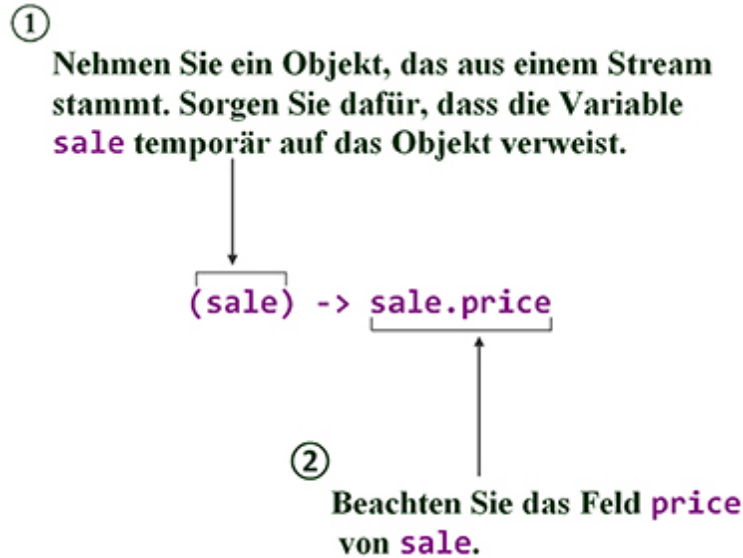
In den 1930ern verwendete der Mathematiker Alonzo Church den griechischen Buchstaben Lamda (λ), um ein bestimmtes mathematisches Gedankengebäude darzustellen, das »auf die Schnelle« erstellt worden ist. Die Idee überlebte die nächsten Jahrzehnte in der Mathematik und der Informationstechnologie. Heute stellt der Begriff *Lambda-Ausdruck* ein Stückchen Code dar, das sowohl als Deklaration einer Methode als auch als Methodenaufruf dient und »auf die Schnelle« erstellt worden ist.

Ein ganz einfacher Lambda-Ausdruck

Hier ein Beispiel eines Lambda-Ausdrucks:

```
(sale) -> sale.price
```

[Abbildung 12.7](#) beschreibt, was der Lambda-Ausdruck bedeutet.



[Abbildung 12.7](#): Sie erhalten von einem bestimmten Verkauf den Preis.

Ein Lambda-Ausdruck ist ein kurz gefasster Weg, um eine Methode zu definieren und aufzurufen, ohne ihr einen Namen zu geben. Der Lambda-Ausdruck in [Abbildung 12.7](#) macht (grob ausgedrückt) das, was auch der folgende Code täte:

```
double getPrice(Sale sale) {  
    return sale.price;  
}  
getPrice(sale);
```

(Denken Sie daran, dass die Klasse `Sale` in [Listing 12.3](#) über keine eigene Methode `getPrice` verfügt.) Der Lambda-Ausdruck in [Abbildung 12.7](#) nimmt Objekte aus einem Strom und ruft auf jedem Objekt eine Methode auf, die `getPrice` ähnelt. Das Ergebnis ist ein neuer Strom – ein Strom aus `double`-Werten.

Ein interessanteres Beispiel

Hier ein anderer Lambda-Ausdruck:

```
(sale) -> sale.item.equals("DVD")
```

[Abbildung 12.8](#) beschreibt, was dieser Lambda-Ausdruck bedeutet.

①

Nehmen Sie ein Objekt, das aus einem Stream stammt. Sorgen Sie dafür, dass die Variable **sale** temporär auf das Objekt verweist.

↓
(sale) -> sale.item.equals("DVD")

②

↑
Beachten Sie das Feld **item**.

③

↑
Ist es **true** oder **false**, dass das Feld **item** von **Sale** gleich "DVD" ist?

[Abbildung 12.8](#): Ist das Objekt, das verkauft wird, eine DVD?

Der Lambda-Ausdruck in [Abbildung 12.8](#) macht (grob ausgedrückt) das, was auch der folgende Code täte:

```
boolean itemIsDVD(Sale sale) {  
    if sale.item.equals("DVD") {  
        return true;  
    } else {  
        return false;  
    }  
}  
itemIsDVD(sale);
```

Der Lambda-Ausdruck in [Abbildung 12.8](#) nimmt sich Objekte aus einem Strom und ruft auf jedem Objekt eine Methode auf, die `itemIsDVD` ähnelt. Das Ergebnis besteht aus `boolean`-Werten – `true` für jeden Verkauf einer DVD und `false` für den Verkauf anderer Artikel.



Sie können die Methode `itemIsDVD` mit oder ohne Lambda-Ausdruck als einzeiligen Körper neu schreiben:

```
boolean itemIsDVD(Sale sale) {
    return sale.item.equals("DVD");
}
```

Ein Lambda-Ausdruck mit zwei Parametern

Schauen Sie sich diesen Lambda-Ausdruck an:

```
(price1, price2) -> price1 + price2
```

[Abbildung 12.9](#) beschreibt die Bedeutung dieses neuen Lambda-Ausdrucks.

①

Nehmen Sie zwei Werte, die aus einem Stream stammen. Sorgen Sie dafür, dass die Variablen **price1** und **price2** temporär auf diese Werte verweisen.

(price1, price2) -> price1 + price2

②

Bilden Sie die Summe der beiden Werte.

[Abbildung 12.9](#): Fügen Sie zwei Preise hinzu.

Der Lambda-Ausdruck in [Abbildung 12.9](#) macht (grob ausgedrückt) das, was auch der folgende Code täte:

```
double sum(double price1, double price2) {
    return price1 + price2;
}
sum(price1, price2);
```

Der Lambda-Ausdruck in [Abbildung 12.9](#) nimmt einen Wert aus einem Strom und ruft eine Methode auf, die `sum` ähnelt und beide Werte addiert. Das Ergebnis ist die Summe aller Preise.

Das schwarze Schaf der Lambda-Ausdrücke

Hier ein sehr interessanter Lambda-Ausdruck:

```
(sale) -> System.out.println(sale.price)
```

Dieser Lambda-Ausdruck macht (grob ausgedrückt) das, was auch der folgende Code täte:

```
void display(Sale sale) {  
    System.out.println(sale.price);  
}  
display(sale);
```

Der Lambda-Ausdruck nimmt sich Objekte aus einem Strom und ruft auf jedem Objekt eine Methode auf, die `display` ähnelt. Im Kopf der Methode `display` weist das Wort `void` darauf hin, dass die Methode keinen Wert zurückgibt. Wenn Sie die Methode `display` aufrufen (oder ihre Entsprechung als Lambda-Ausdruck verwenden), erwarten Sie eigentlich auch keinen Rückgabewert. Stattdessen rechnen Sie damit, dass der Code etwas als Reaktion auf den Aufruf macht (zum Beispiel Text auf dem Bildschirm des Computers anzeigt).

Um eine genaue Trennlinie zwischen der Rückgabe eines Wertes und »Etwas machen« zu ziehen, haben funktionale Programmierer einen Namen dafür, wenn etwas ohne Rückgabe eines Wertes getan wird. Sie nennen es *Seiteneffekt*. In der funktionalen Programmierung wird ein Seiteneffekt als Bürger zweiter Klasse, als letzter Ausweg oder als eine Taktik betrachtet, auf die Sie zurückgreifen, wenn Sie nicht einfach ein Ergebnis zurückgeben können. Komischerweise wird auch die Anzeige von Daten auf dem Bildschirm (was eigentlich eine der Aufgaben vieler Programme ist) als Seiteneffekt betrachtet. Jedes Programm, das eine Ausgabe (auf einem Bildschirm, auf Papier oder als Kaffeesatz) anzeigt, ist ein funktionales Programm.

Eine Klassifizierung von Lambda-Ausdrücken

Java 8 teilt Lambda-Ausdrücke in ungefähr 45 verschiedene Kategorien ein. [Tabelle 12.2](#) listet ein paar davon auf.

Name	Beschreibung	Beispiel

Name	Beschreibung	Beispiel
Function	Akzeptiert einen Parameter; liefert ein Ergebnis eines beliebigen Typs	<code>(sale) -> sale</code>
Predicate	Akzeptiert einen Parameter; liefert einen <code>boolean</code> -Wert	<code>(sale) -> sale.item</code>
BinaryOperator	Akzeptiert zwei Parameter desselben Typs; liefert ein Ergebnis desselben Typs	<code>(price1, price2) -> price1 + price2</code>
Consumer	Akzeptiert einen Parameter; liefert kein Ergebnis	<code>(sale) -> System.out.println(sale.price)</code>

Tabelle 12.2: Lambda-Ausdrücke



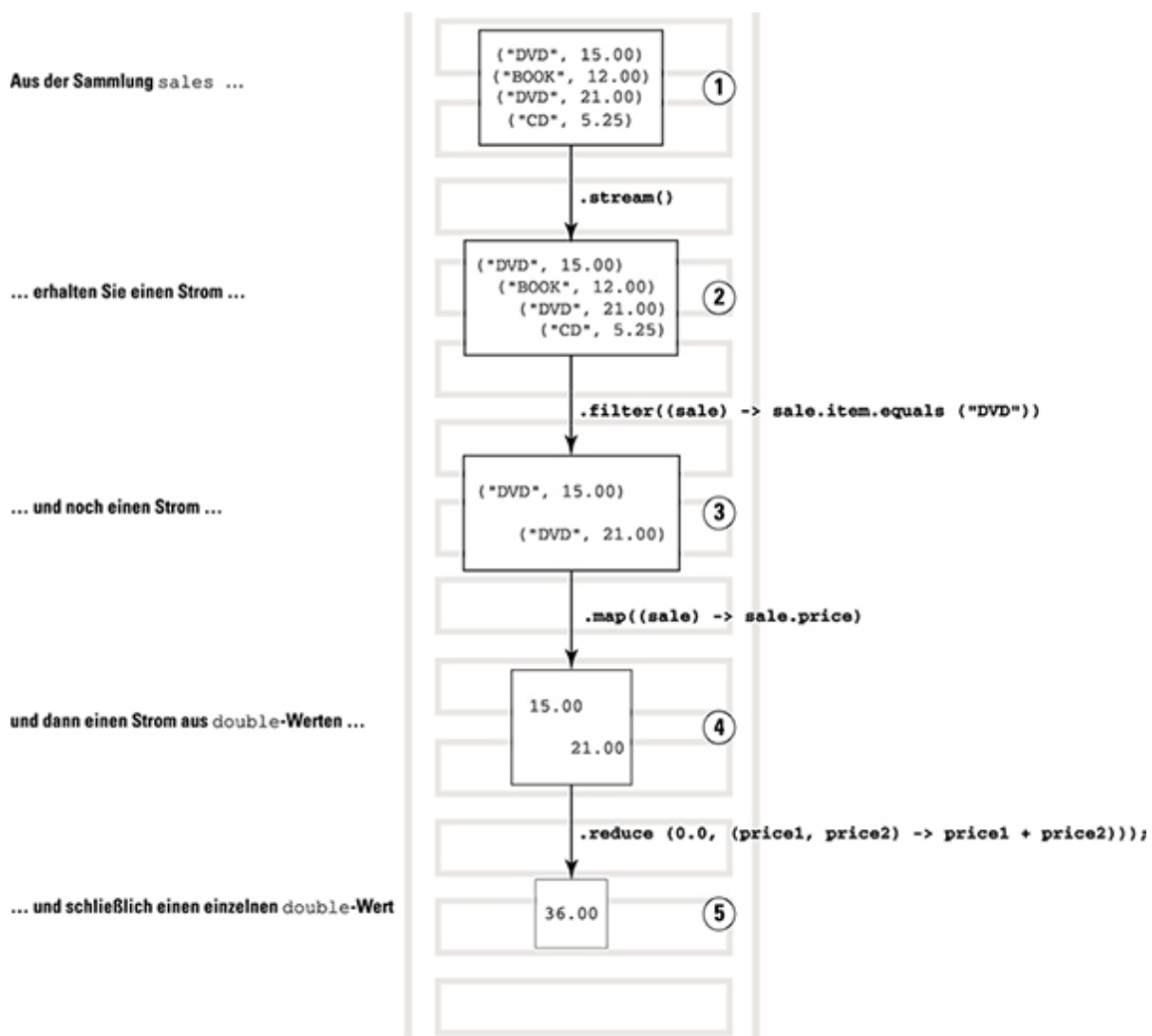
Die Kategorien in [Tabelle 12.2](#) schließen sich gegenseitig nicht aus. So ist zum Beispiel jedes `Predicate` auch eine `Function`. (Jedes `Predicate` akzeptiert einen Parameter und gibt ein Ergebnis zurück. Dieses Ergebnis ist zufällig ein Wert vom Typ `boolean`.)

Streams und Lambda-Ausdrücke verwenden

Java 8 enthält wirklich gute Methoden, die Streams und Lambda-Ausdrücke optimal nutzen. Sie können mit Streams und Lambda-Ausdrücken eine sogenannte Assembly Line erstellen, die die Verkaufsprobleme dieses Kapitels elegant löst. Die neue Lösung

verwendet, anders als der Code in [Listing 12.4](#), funktionale Programmierung.

Sie können sich eine Assembly Line als Fertigungsstraße mit einem Fließband vorstellen. Die *Assembly Line* hier besteht aus mehreren Methoden. Jede Methode übernimmt Daten, bearbeitet diese Daten irgendwie und übergibt die Ergebnisse der Bearbeitung an die nächste Methode in der Reihe. [Abbildung 12.10](#) stellt die Assembly Line für das Verkaufsproblem dieses Kapitels dar.



[Abbildung 12.10](#): Eine Assembly Line der funktionalen Programmierung

In [Abbildung 12.10](#) stellt jedes Kästchen einen Satz an Rohmaterialien dar, die entlang einer Assembly Line umgewandelt werden. Jeder Pfeil entspricht einer Methode.

So siebt zum Beispiel der Übergang vom zweiten zum dritten Kästchen eine Methode (die Methode `filter`) Verkäufe von Objekten heraus, die keine DVDs sind. Stellen Sie sich hier jemanden vor, der zwischen dem zweiten und dem dritten Kästchen steht und Bücher und CDs aus der Assembly Line entnimmt und einfach auf den Boden fallen lässt.

Bei dem Parameter der Methode `filter` handelt es sich um ein `Predicate` – einen Lambda-Ausdruck mit einem `boolean` Ergebnis (siehe [Tabelle 12.2](#) und [Tabelle 12.3](#)). Die Methode `filter` filtert in [Abbildung 12.10](#) Elemente heraus, die den `true/false`-Test des Lambda-Ausdrucks nicht bestehen.

Methode	Mitglied von	Parameter	Typ des Ergebnisses	Wert des Ergebnisses
<code>stream</code>	<code>Collection</code> (zum Beispiel ein <code>ArrayList</code> -Objekt)	(keiner)	<code>Stream</code>	Ein Strom, der die Elemente einer Sammlung ausgibt
<code>filter</code>	<code>Stream</code>	<code>Predicate</code>	<code>Stream</code>	Ein neuer Strom, der Werte enthält, für die der Lambda-Ausdruck <code>true</code> zurückgibt

Methode	Mitglied von	Parameter	Typ des Ergebnisses	Wert des Ergebnisses
map	Stream	Function	Stream	Ein neuer Strom, der das Ergebnis dessen enthält, was die Anwendung eines Lambda-Ausdrucks auf den eingehenden Strom als Ergebnis liefert
reduce	Stream	Binary Operator	Der Typ, den der Binary Operator verwendet	Das Ergebnis einer Kombination aller Werte eines eingehenden Stroms

Tabelle 12.3: Einige Methoden der funktionalen Programmierung

In [Abbildung 12.10](#) entnimmt eine Methode (die Methode `map`) dem einzelnen Verkauf (`sale`) zwischen dem dritten und dem vierten Kästchen den Preis (`price`). Von diesem Punkt an enthält die Assembly Line nur noch `price`-Werte.

Um genau zu sein, muss es eigentlich heißen, dass Javas Methode `map` eine `Function` wie

```
(sale) -> sale.price
```

nimmt und auf jeden Wert in einem Strom anwendet (siehe [Tabelle 12.2](#) und [Tabelle 12.3](#)). Die Methode `map` nimmt sich also in [Abbildung 12.10](#) einen eingehenden Strom aus `sale`-Objekten und erstellt einen abgehenden Strom aus `price`-Werten.

In [Abbildung 12.10](#) summiert eine Methode (die Methode `reduce`) beim Übergang vom vierten zum fünften Kästchen die Preise der DVD-Verkäufe.

Java's Methode `reduce` verfügt über zwei Parameter:

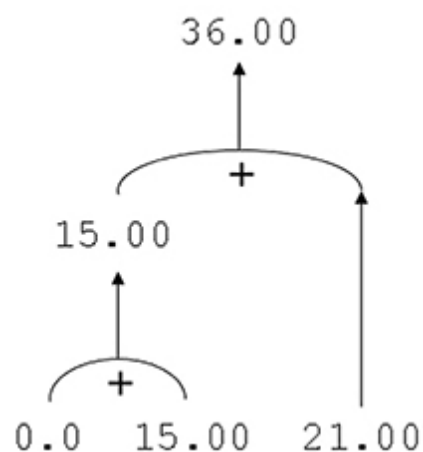
- ✓ Der erste Parameter ist ein Initialisierungswert.
In [Abbildung 12.10](#) beträgt der Initialisierungswert `0.0`.

- ✓ Der zweite Parameter ist ein `BinaryOperator` (siehe [Tabelle 12.2](#) und [Tabelle 12.3](#)).

In [Abbildung 12.10](#) lautet der `BinaryOperator` der Methode `reduce`

`(price1, price2) -> price1 + price2`

Die Methode `reduce` verwendet ihren `BinaryOperator`, um die Werte im ankommenden Strom zu kombinieren. Der Initialisierungswert dient der gesamten Operation als Startpunkt. In [Abbildung 12.10](#) führt die Methode `reduce` zwei Additionen durch (siehe [Abbildung 12.11](#)).

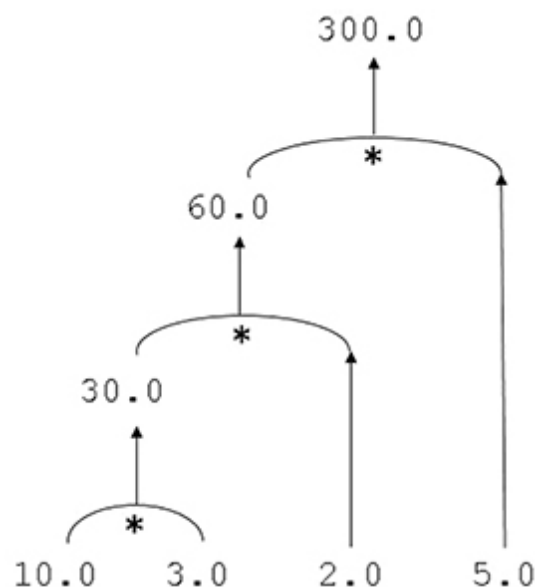


[Abbildung 12.11](#): Die Methode *reduce* addiert zwei Werte eines ankommenden Stroms.

Stellen Sie sich zum Vergleich vor, dass Sie mit dem Strom die Methode

```
reduce(10.0, (value1, value2) -> value1 * value2)
```

aufrufen, wobei der Strom unter anderem die Werte 3.0, 2.0 und 5.0 enthält. [Abbildung 12.12](#) stellt die daraus resultierende Aktion dar.



[Abbildung 12.12](#): Die Methode *reduce* multipliziert Werte eines ankommenden Stroms.



Vielleicht haben Sie schon von Googles Programmiermodell *MapReduce* gehört. Die Ähnlichkeit zwischen dem Namen des Programmiermodells und den Namen der Java-Methoden `map` und `reduce` ist kein Zufall.

Wenn Sie die Assembly Line aus [Abbildung 12.10](#) als Ganzes betrachten, summiert sie die Preise von verkauften DVDs. [Listing 12.5](#) enthält ein vollständiges Programm, das die Streams und Lambda-Ausdrücke aus [Abbildung 12.10](#) verwendet.

```
import java.text.NumberFormat;
```

```

import java.util.ArrayList;
public class TallySales {
    public static void main(String[] args) {
        ArrayList<Sale> sales = new ArrayList<Sale>
();
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();
        fillTheList(sales);
        System.out.println(currency.format(
sales.stream()
.filter((sale) -
> sale.item.equals("DVD"))
.map((sale) -> sale.price)
.reduce
(0.0, (price1, price2) -
> price1 + price2)));
    }

    static void fillTheList(ArrayList<Sale> sales) {
        sales.add(new Sale("DVD", 15.00));
        sales.add(new Sale("Book", 12.00));
        sales.add(new Sale("DVD", 21.00));
        sales.add(new Sale("CD", 5.25));
    }
}

```

[Listing 12.5](#): Den funktionalen »Way of Life« leben

Bei dem in [Listing 12.5](#) fett gedruckten Code handelt es sich um einen großen Java-Befehl. Dieser Befehl besteht aus einer Folge von Methodenaufrufen. Jeder Methodenaufruf gibt ein Objekt zurück und jedes dieser Objekte ist das Element vor dem Punkt des nächsten Methodenaufrufs. Damit wissen Sie, wie Sie eine Assembly Line formulieren.

So verweist zum Beispiel zu Beginn des fett gedruckten Codes der Name `sale` auf ein `ArrayList`-Objekt. Jedes `ArrayList`-Objekt verfügt über eine `stream`-Methode. Damit ist in [Listing 12.5](#) `sales.stream()` ein Aufruf der Methode `stream` des `ArrayList`-Objekts.

Die Methode `stream` gibt eine Instanz der Java-Klasse `Stream` zurück. (Welch eine Überraschung!) Auf diese Weise verweist

`sales.stream` auf ein `Stream`-Objekt (siehe [Abbildung 12.13](#)).

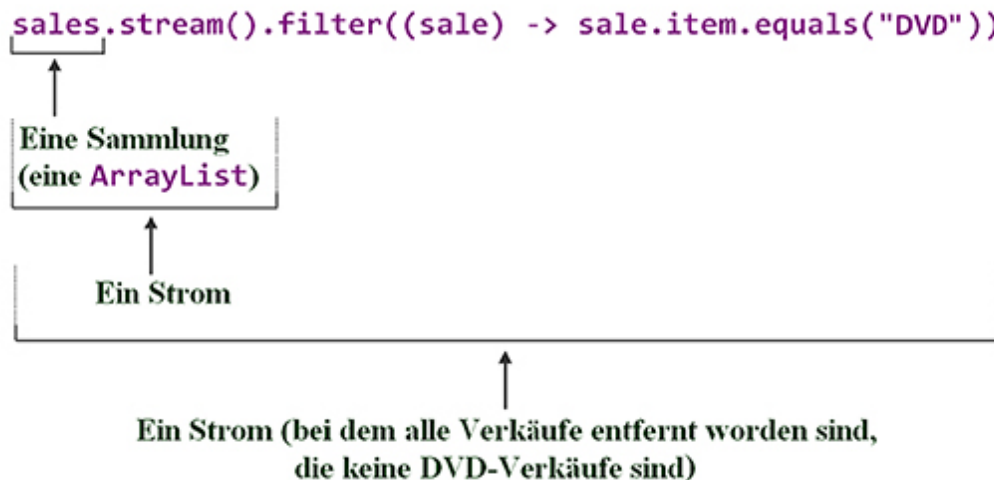
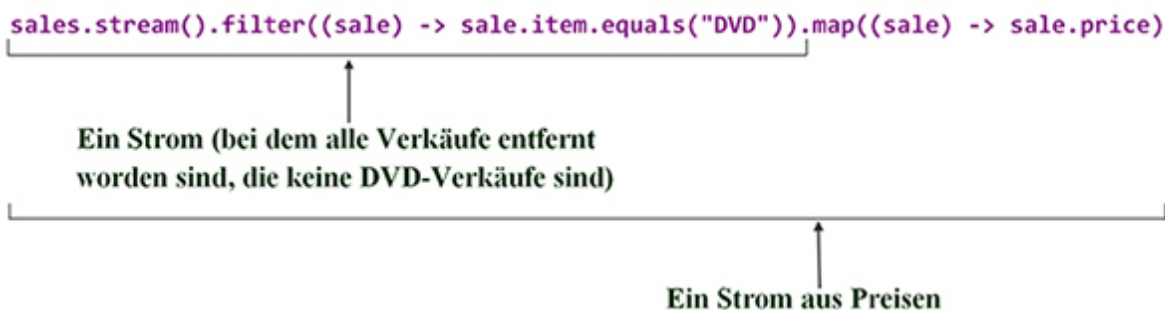


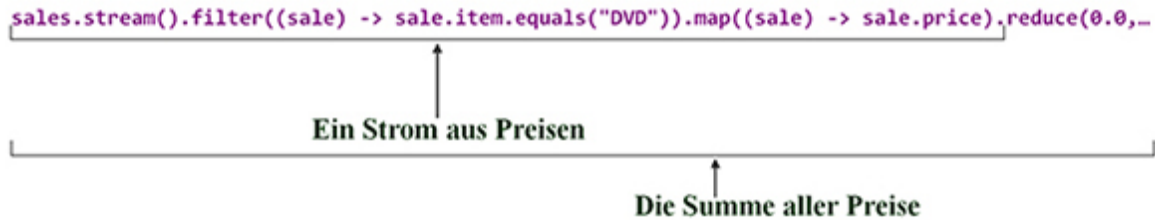
Abbildung 12.13: Funktionalen Code von innen nach außen lesen

Jedes Stream-Objekt verfügt über eine Methode `filter`. Damit ist `sales.stream().filter((sale) -> sale.item.equals("DVD"))` ein Aufruf der Methode `filter` des Objekts `Stream` (siehe [Abbildung 12.13](#)).

Dieses Muster setzt sich fort. Die Methode `map` des Objekts `Stream` gibt ebenfalls ein Objekt `Stream` zurück – ein Objekt, das Preise enthält (siehe [Abbildung 12.14](#)). Sie wenden nun auf diesen Strom aus Preisen die Methode `reduce` an, die einen `double`-Wert ausgibt – die Summe der DVD-Verkäufe (siehe [Abbildung 12.15](#)).



[Abbildung 12.14](#): Den Preis jeder DVD erhalten



[Abbildung 12.15](#): Die Summer aller Preise erhalten

Warum sich Sorgen machen?

Die Kette der Methodenaufrufe in [Listing 12.5](#) erledigt alles, was auch die Schleife in [Listing 12.4](#) macht. Aber der Code in [Abbildung 12.15](#) verwendet Konzepte aus der funktionalen Programmierung. Was ist denn dann das Besondere? Sind Sie mit [Listing 12.5](#) besser dran als mit Listing 12.4?

Sie sind es. In den vergangenen Jahren ging der Trend beim Design von Chips zu den Multicoreprozessoren. Ein Prozessor, der über mehrere Kerne verfügt, ist in der Lage, mehrere Befehle gleichzeitig auszuführen. Dies beschleunigt die Programmausführung um den Faktor 2, 4, 8 oder noch mehr. Programme laufen viel schneller, wenn Sie die Arbeit auf mehrere Kerne verteilen. Aber wie machen Sie so etwas?

Sie können den imperativen Code in [Listing 12.4](#) modifizieren. So sind Sie zum Beispiel in der Lage, durch den Einbau weniger Funktionen Schleifendurchgänge an mehrere Kerne zu übergeben. So etwas führt aber meistens zu einem Code, der ziemlich chaotisch ist. Damit der Code sauber arbeitet, müssen Sie die Schleifendurchgänge in allen Einzelheiten verwalten und dabei sorgfältig darauf achten, dass das Endergebnis auch richtig ist.

Im Gegensatz dazu lässt sich funktionaler Code leicht anpassen. Um Multicoreprozessoren nutzen zu können, ändern Sie in [Listing 12.5](#) *nur ein einziges Wort!*

```
sales.parallelStream()
    .filter((sale) -> sale.item.equals("DVD"))
    .map((sale) -> sale.price)
    .reduce
        (0.0, (price1, price2) -> price1 + price2)
```

Ein paralleler Strom (*Parallel Stream*) ist wie ein normaler Strom, nur dass die Objekte nicht in einer bestimmten Reihenfolge verarbeitet werden. Die Werte eines parallelen Stroms hängen nicht voneinander ab, was es Ihnen ermöglicht, mehrere Werte gleichzeitig zu verarbeiten. Wenn Ihr Prozessor über mehr als einen Kern verfügt (was heutzutage eigentlich die Regel ist), können Sie die Werte im Strom unter den verfügbaren Kernen aufteilen. Jeder Kern arbeitet unabhängig von den anderen Kernen und jeder Kern übergibt seine Ergebnisse an eine letzte `reduce`-Methode. Diese Methode fasst die Ergebnisse der Kerne in einem abschließenden Zähler zusammen und Ihre Antwort ist blitzschnell berechnet worden.

Methodenreferenzen

Werfen Sie einen kritischen Blick auf den letzten Lambda-Ausdruck in [Listing 12.5](#):

```
(price1, price2) -> price1 + price2)
```

Dieser Ausdruck erledigt, grob ausgedrückt, dieselbe Arbeit wie eine `sum`-Methode. (Sie finden im Abschnitt *Lambda-Ausdrücke* die Deklaration einer solchen Methode `sum`.) Wenn Sie vor der Wahl stehen, entweder eine dreizeilige `sum`-Methode oder einen einzeiligen Lambda-Ausdruck einzugeben, entscheiden Sie sich wohl für den Lambda-Ausdruck. Was ist aber, wenn es eine dritte Alternative gäbe? Statt eine eigene Methode `sum` zu schreiben, können Sie auf eine vorhandene Methode `sum` zugreifen. Und dies wäre dann auch der schnellste und sicherste Weg, den es einzuschlagen gilt.

Keine Variablen? Kein Problem!

Bedenken Sie noch einmal die Probleme, auf die ich zu Beginn des Abschnitts *Neu in Java 8: funktionale Programmierung* hinweise. Mehrere Clients versuchen gleichzeitig, Oracles Aktienkurs zu aktualisieren, oder zwei

Besucher versuchen gleichzeitig, auf einer Website dasselbe Objekt zu kaufen. Die Quelle des Problems sind die gemeinsam genutzten Daten. Wie viele Clients teilen sich den Zugriff auf Oracles Aktienkurs? Wie viele Kunden teilen sich den Zugriff auf die Schaltfläche KAUFEN einer Webseite? Wie viele Kerne Ihres Prozessors können den Wert derselben Variablen ändern? Wenn Sie das Problem der gemeinsamen Datennutzung gelöst haben, sind Sie auch das Problem der Multicoreverarbeitung los.

Bei der imperativen Programmierung ist eine Variable ein Ort, an dem Befehle ihre Werte miteinander teilen. Besteht nun eine Möglichkeit, dass Sie in Ihrem Code auf den Einsatz von Variablen verzichten können?

Vergleichen Sie die Schleife in [Listing 12.4](#) mit dem Code der funktionalen Programmierung in Listing 12.5. In [Listing 12.4](#) wird die Variable `total` von allen Schleifeniterationen geteilt. Da jeder Schleifendurchlauf den Wert von `total` ändern kann, dürfen Sie auf keinen Fall jeden Schleifendurchlauf einem anderen Prozessorkern zuweisen. Falls Sie so etwas doch machen, riskieren Sie es, dass zwei Kerne `total` gleichzeitig aktualisieren. (Dieses Risiko besteht, weil aufgrund der gleichzeitigen Aktualisierung keiner der Kerne seine Aufgabe fehlerfrei ausführt!) Nun kennt der Code der funktionalen Programmierung (in Listing 12.5) keine Variable `total`. Hier spielt also ein laufendes `total` keine Rolle. Stattdessen summiert in [Listing 12.5](#) die Methode `reduce` Werte, die aus einem Strom stammen. Dieser ankommende Strom entstammt dem Aufruf der vorherigen Methode (der Methode `map`), weshalb dieser Strom keinen Namen hat. Das ist doch gut so, oder? Sie benötigen keine Variable, um einen Strom aus Werten zu speichern.

Bei der imperativen Programmierung ist eine Variable ein Ort, an dem Befehle ihre Werte teilen. Die funktionale Programmierung vermeidet Variablen. Wenn Sie also funktional programmieren, müssen Sie sich nicht so viel um geteilte Daten kümmern. Viele Probleme, die mit Multicorprozessoren zusammenhängen, lösen sich in Luft auf. Ihr Code kann die Vorteile mehrerer Kerne nutzen. Wenn Sie den Code schreiben, müssen Sie sich nicht um Daten kümmern, die unter den Kernen aufgeteilt werden. Damit stehen Sie vor einer eleganten Lösung eines der zentralen Probleme des Computerwesens.

Wie es das Schicksal so will, gibt es in Java 8 eine Klasse mit dem Namen `Double`. Diese Klasse enthält eine statische Methode `sum`. Sie müssen also keine eigene Methode `sum` erstellen. Wenn Sie den Code

```
double i = 5.0, j = 7.0;  
System.out.println(Double.sum(i, j));
```

ausführen, zeigt der Computer 12.0 an. Anstatt in [Listing 12.5](#) den Lambda-Ausdruck `price1 + price2` zu schreiben, können Sie eine Methodenreferenz erstellen.

```
sales.stream()
    .filter((sale) -> sale.item.equals("DVD"))
    .map((sale) -> sale.price)
    .reduce(0.0, Double::sum)
```

Der Ausdruck `Double::sum` verweist auf die Methode `sum`, die zu Javas Klasse `Double` gehört. Wenn Sie die Methodenreferenz `Double::sum` verwenden, erreichen Sie dasselbe wie mit dem letzten Lambda-Ausdruck in [Listing 12.5](#).

Sie finden in [Kapitel 10](#) Informationen über statische Methoden, und die Java-Klasse `Double` ähnelt in vielen Dingen der Klasse `Integer`, die ich in [Kapitel 11](#) beschreibe. Die Klasse `Double` enthält Methoden und andere Werkzeuge, die mit `double`-Werten umgehen können.

13

Auch dann noch gut aussehen, wenn die Dinge eine unerwartete Wendung nehmen

In diesem Kapitel ...

- ▶ Sich von falschen Eingaben erholen und aus anderen üblen Situationen retten
 - ▶ Den Code (mehr oder weniger) absturzsicher machen
 - ▶ Eigene Exception-Klassen definieren
-

9. September 1945: Eine Motte fliegt in eines der Relais des Harvard-Computers Mark II und vermasselt alles. Dies ist der erste echte *Bug* eines Computers, der dokumentiert wurde.

19. April 1957: Herbert Bright, Leiter des Zentrums für Datenverarbeitung bei Westinghouse in Pittsburgh, erhält per Post einen nicht gekennzeichneten Satz Lochkarten (was Sie heutzutage mit einer unbeschrifteten CD-ROM vergleichen können). Mr. Bright geht davon aus, dass dieser Stapel vom FORTRAN-Entwicklungsteam kommt. (FORTRAN war die erste Programmiersprache für Computer.) Er hat seit Jahren auf diese Software gewartet. (Damals gab es noch kein Herunterladen aus dem Internet.)

Nur mit seinem Vertrauen gewappnet, schreibt er ein kleines FORTRAN-Programm und versucht, es auf seinem IBM 704 zu kompilieren. (Der IBM 704 steht in seinem eigenen, extra für ihn gebauten, fast 200 Quadratmeter großen Raum. Er hat

Vakuumröhren und keine Transistoren und verfügt über den Wahnsinnsarbeitsspeicher von 32 K. Das Betriebssystem muss von Bändern geladen werden, bevor ein Programm ausgeführt werden kann. Eine ganz normale Programmausführung dauert zwischen zwei und vier Stunden.) Nach der üblichen Wartezeit liefert Brights Versuch, ein FORTRAN-Programm zu kompilieren, nur eine Fehlermeldung, denn in einer Anweisung fehlt ein Komma. Bright korrigiert seinen Fehler und das Programm funktioniert wunderbar.

22. Juli 1962: Mariner I, die als erste US-Raumsonde einen anderen Planeten erreichen soll, wird vier Minuten nach dem Start zerstört, weil sie außer Kontrolle gerät. Das Verhalten der Raumsonde wird mit einem fehlenden Strich (vergleichbar einem Bindestrich) in der Formel für die Geschwindigkeit der Rakete erklärt.

Zur selben Zeit wird in der Software der NASA zur Berechnung von Umlaufbahnen der falsche Befehl `DO 10 I=1.10` gefunden. (Es hätte `DO 10 I=1,10` heißen müssen.) In moderner Schreibweise ist das so, als wenn Sie `do10i = 1.10` statt `for (int i=1; i<=10; i++)` schreiben. Die Änderung von einem Komma zu einem Punkt macht aus einer Schleife einen Zuweisungsbefehl.

1. Januar 2000: Das Jahr-2000-Problem treibt in der modernen Welt sein Unwesen.

Mit Ausnahmen umgehen

Sie machen Inventur. Das heißt Zählen – Teil für Teil, Karton für Karton – und Aufschreiben der Zahlen auf großen Papierbögen, in kleinen Handhelds oder mit der Tastatur des Computers direkt in die entsprechenden Formulare. Ein wichtiger Teil dieses Projekts ist die Eingabe der Anzahl an Kartons, die sich im Regal »Die großen staubigen Kartons, die seit Bestehen des Unternehmens nicht geöffnet worden sind« befinden. Und auch Sie brechen die Jahrzehnte alte Gewohnheit nicht und entscheiden, keinen dieser

Kartons aufzumachen. Sie legen einfach fest, dass jeder Karton 3,25 Euro wert ist.

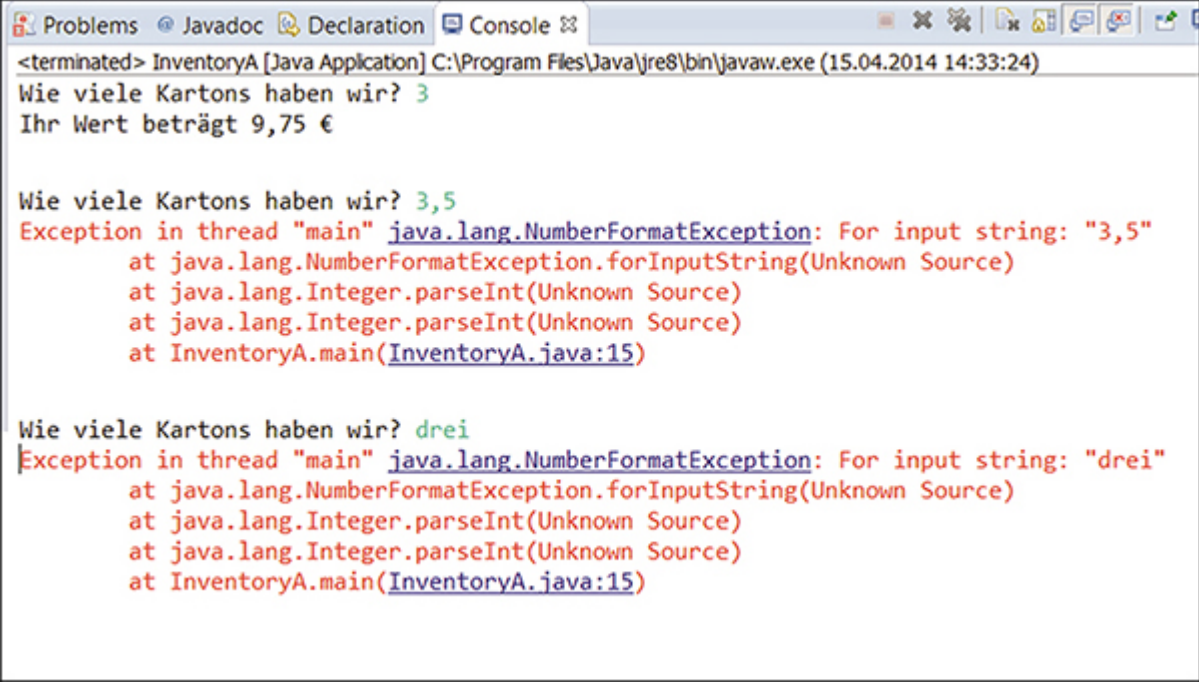
[Listing 13.1](#) zeigt die Software, die dieses Stückchen der Inventur übernimmt. Die Software weist eine Schwachstelle auf, die [Abbildung 13.1](#) darstellt. Wenn der Benutzer eine Ganzzahl eingibt, ist alles in Ordnung. Wenn der Benutzer aber etwas anderes (wie zum Beispiel die Zahl 3, 5) eingibt, stürzt das Programm ab. Natürlich kann dagegen etwas unternommen werden. Ein Computer ist dumm, aber er ist nicht so dumm, dass er vollständig versagt, wenn ein Benutzer einen falschen Wert eingibt.

```
import static java.lang.System.out;
import java.util.Scanner;
import java.text.NumberFormat;
public class InventoryA {
    public static void main(String args[]) {
        final double boxPrice = 3.25;
        Scanner keyboard = new Scanner(System.in);
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();
        out.print("Wie viele Kartons haben wir? ");
        String numBoxesIn = keyboard.next();
        int numBoxes = Integer.parseInt(numBoxesIn);
        out.print("Ihr Wert beträgt ");
        out.println(currency.format(numBoxes * boxPrice
));
        keyboard.close();
    }
}
```

[Listing 13.1](#): Kartons zählen

Der Schlüssel zum Beheben eines Programmfehlers liegt im Untersuchen der Meldung, die erscheint, wenn das Programm abstürzt. Die Meldung des Inventurprogramms lautet `java.lang.NumberFormatException`. Dies bedeutet, dass es im API-Paket `java.lang` eine Klasse mit dem Namen `NumberFormatException` gibt. Irgendwie hat der Aufruf von `Integer.parseInt` diese Klasse aus ihrem Versteck geholt.

(Sie finden in [Kapitel 11](#) eine kurze Beschreibung der Methode `Integer.parseInt()`.)



```
<terminated> InventoryA [Java Application] C:\Program Files\Java\jre8\bin\javaw.exe (15.04.2014 14:33:24)
Wie viele Kartons haben wir? 3
Ihr Wert beträgt 9,75 €

Wie viele Kartons haben wir? 3,5
Exception in thread "main" java.lang.NumberFormatException: For input string: "3,5"
    at java.lang.NumberFormatException.forInputString(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at InventoryA.main(InventoryA.java:15)

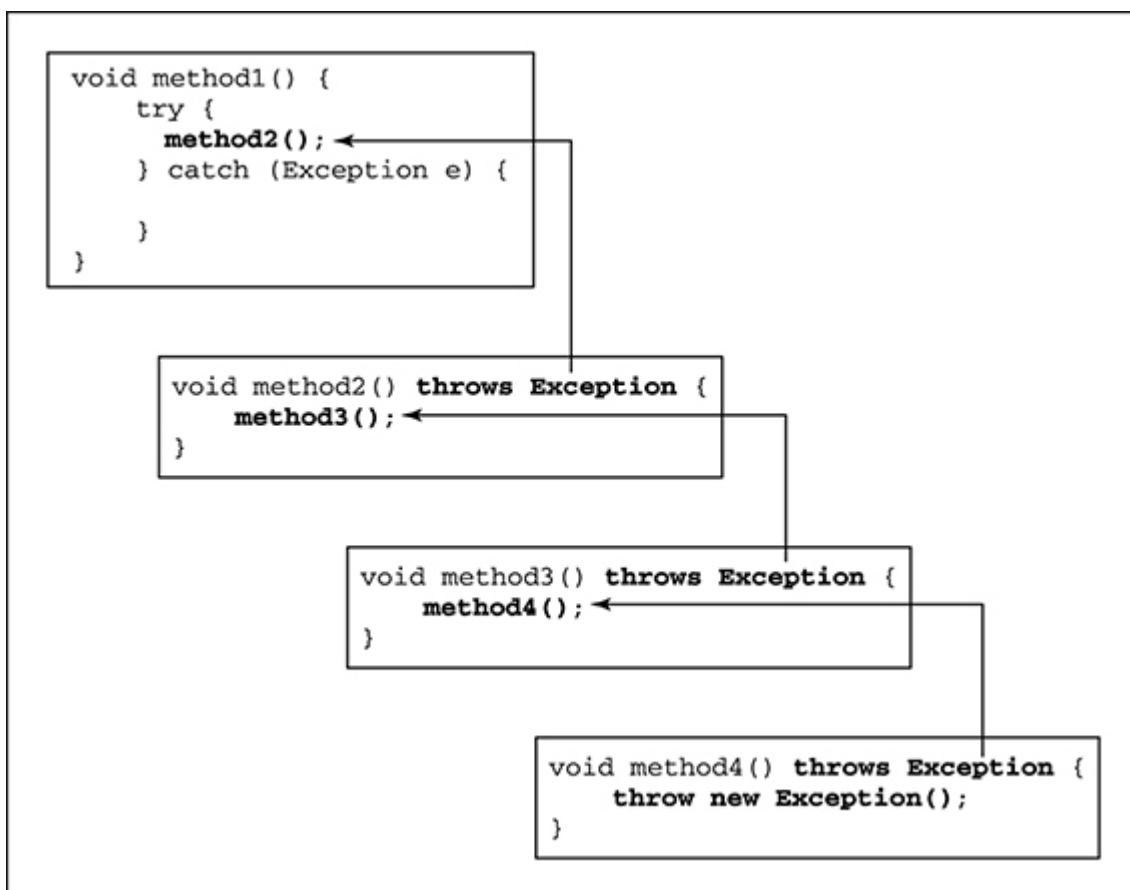
Wie viele Kartons haben wir? drei
Exception in thread "main" java.lang.NumberFormatException: For input string: "drei"
    at java.lang.NumberFormatException.forInputString(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at InventoryA.main(InventoryA.java:15)
```

[Abbildung 13.1](#): Drei Ausführungen von [Listing 13.1](#)

Nun, das hier geschieht: Die Programmiersprache Java kennt einen Mechanismus, der *Exception Handling* oder auf Deutsch *Ausnahmebehandlung* heißt. Mit der Ausnahmebehandlung ist ein Programm in der Lage, die Dinge zu entdecken, die schief laufen, und zu reagieren, indem ein ganz neues Objekt erstellt wird. In der offiziellen Terminologie sagt man in solch einem Fall, dass das Programm eine Ausnahme *wirft*. Das neue Objekt, das eine Instanz der Klasse `Exception` ist, wird so lange wie eine heiße Kartoffel von einem Teil des Codes an einen anderen Teil weitergegeben, bis sich ein Teil des Codes entscheidet, die Ausnahme *zu fangen*. Wenn die Ausnahme gefangen worden ist, führt das Programm einen Wiederherstellungscode aus, beerdigt die Ausnahme und macht so mit dem nächsten normalen Befehl weiter, als wenn nichts gewesen wäre. [Abbildung 13.2](#) stellt diesen Vorgang grafisch dar.

Die ganze Sache wird mithilfe einiger Java-Schlüsselwörter erledigt. Bei diesen Schlüsselwörtern handelt es sich um:

- ✓ **throw:** (deutsch *werfen*) Erstellt ein neues Ausnahmeobjekt.
- ✓ **throws:** Gibt den Schwarzen Peter von einer Methode an den Code weiter, der die Methode aufgerufen hat.
- ✓ **try:** Hierzu gehört Code, der das Potenzial besitzt, ein neues Ausnahmeobjekt zu erstellen. Im Allgemeinen enthält der Code in einer `try`-Klausel Aufrufe von Methoden, deren Code eine oder mehrere Ausnahmen erstellen können.
- ✓ **catch:** Kümmt sich um die Ausnahme, beendet sie und macht mit dem Programmablauf weiter.



[Abbildung 13.2](#): Eine Ausnahme werfen, weitergeben und fangen

Damit kennen Sie die Wahrheit. Die Methode `Integer.parseInt` kann aufgrund einer Ereigniskette wie der in [Abbildung 13.2](#) eine `NumberFormatException` werfen. Wenn Sie `Integer.parseInt` aufrufen, wird diese `NumberFormatException` an Sie übergeben.



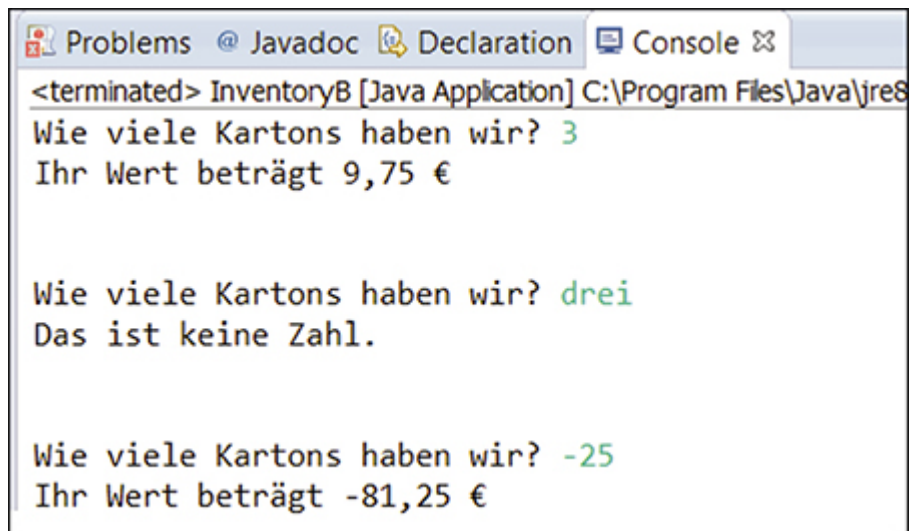
Die Dokumentation der Java-API für die Methode `parseInt` besagt in freier Übersetzung: »Throws: `NumberFormatException` – wenn der String keinen Integer-Wert enthält, der geparkt werden kann.« Es lohnt sich wirklich, ab und zu einen Blick in diese Dokumentation zu werfen.

Wenn Sie sich selbst zu den Helden zählen, sollten Sie die Ausnahme fangen, damit der restliche Code mit seiner normalen Arbeit weitermachen kann. [Listing 13.2](#) zeigt, wie eine Ausnahme gefangen werden kann.

```
import static java.lang.System.out;
import java.util.Scanner;
import java.text.NumberFormat;
public class InventoryB {
    public static void main(String args[]) {
        final double boxPrice = 3.25;
        Scanner keyboard = new Scanner(System.in);
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();
        out.print("Wie viele Kartons haben wir? ");
        String numBoxesIn = keyboard.next();
        try {
            int numBoxes = Integer.parseInt(numBoxesIn
);
            out.print("Ihr Wert beträgt ");
            out.println(
                currency.format(numBoxes * boxPrice));
        } catch (NumberFormatException e) {
            out.println("Das ist keine Zahl.");
        }
        keyboard.close();
    }
}
```

[Listing 13.2](#): Ein Held zählt Kartons.

[Abbildung 13.3](#) gibt das Ergebnis von drei Ausführungen des Codes in [Listing 13.2](#) wieder. Wenn ein irregeleiteter Benutzer drei statt 3 eingibt, bleibt das Programm ruhig und zeigt einfach `Das ist keine Zahl.` an. Der Trick besteht darin, den Aufruf von `Integer.parseInt` in einer `try`-Klausel unterzubringen. Wenn Sie dies machen, achtet der Computer auf Ausnahmen, wenn ein Befehl in der `try`-Klausel ausgeführt wird. Wenn dann eine Ausnahme geworfen wird, springt der Computer aus der `try`-Klausel heraus in die darunter liegende `catch`-Klausel. In [Listing 13.2](#) ist dies die Klausel `catch (NumberFormatException e)`. Er führt nun den `println`-Befehl in der Klausel aus und macht dann mit der normalen Verarbeitung weiter. (Wenn es in [Listing 13.2](#) im Anschluss an die `catch`-Klausel Befehle gäbe, würde der Computer sie ausführen.)



```
<terminated> InventoryB [Java Application] C:\Program Files\Java\jre8
Wie viele Kartons haben wir? 3
Ihr Wert beträgt 9,75 €

Wie viele Kartons haben wir? drei
Das ist keine Zahl.

Wie viele Kartons haben wir? -25
Ihr Wert beträgt -81,25 €
```

[Abbildung 13.3](#): Drei Ausführungen des Codes in Listing 13.2



Eine vollständiges `try-catch`-Gruppe – eine `try`-Klausel, eine `catch`-Klausel und was Sie sonst noch haben – wird `try`-Befehl genannt. Um den Schwerpunkt besser

hervorzuheben, bezeichne ich so etwas gerne als `try-catch`-Befehl.

Der Parameter einer »catch«-Klausel

Werfen Sie in [Listing 13.2](#) einen Blick auf die `catch`-Klausel und beachten Sie ganz besonders die Wörter

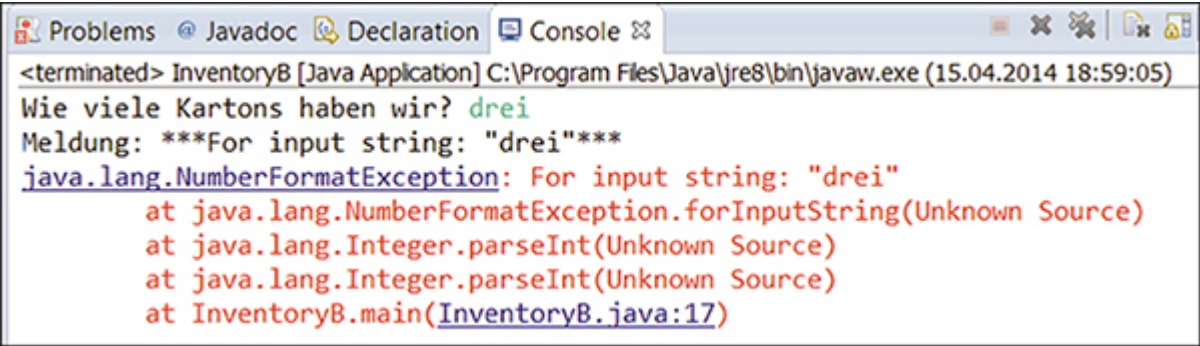
`(NumberFormatException e)`. Dies sieht wie die Parameterliste einer Methode aus. Tatsächlich hat jede `catch`-Klausel etwas von einer Methode mit einer eigenen Parameterliste an sich. Diese Parameterliste enthält immer den Namen eines Ausnahmetyps und einen Parameter.

Ich mache in [Listing 13.2](#) nichts mit dem Parameter `e` der `catch`-Klausel, was aber problemlos möglich wäre. Denken Sie daran, dass die Ausnahme, die geworfen wird, ein Objekt ist – eine Instanz der Klasse `NumberFormatException`. Wenn eine Ausnahme gefangen wird, sorgt der Computer dafür, dass der Parameter der Klausel `catch` auf dieses Ausnahmeobjekt verweist. Oder anders ausgedrückt, der Name `e` speichert Informationen über die Ausnahme. Um daraus Nutzen zu ziehen, können Sie einige der Methoden des Ausnahmeobjekts aufrufen, wie zum Beispiel:

```
} catch (NumberFormatException e) {  
    out.println("Message: ***" + e.getMessage() + "***  
*");  
    e.printStackTrace();  
}
```

Mit dieser neuen `catch`-Klausel könnte eine Ausführung des Inventurprogramms wie das aussehen, was [Abbildung 13.4](#) zeigt. Wenn Sie `getMessage` aufrufen, gelangen Sie an Einzelheiten der Ausnahme. (In [Abbildung 13.4](#) lautet ein solches Detail Nachricht: `***For input string: "drei"***`). Wenn Sie `printStackTrace` aufrufen, erhalten Sie weitere Informationen, zu denen insbesondere die Ausgabe der Methoden gehört, die zu dem Zeitpunkt liefen, als die Ausnahme geworfen wurde. (In [Abbildung 13.4](#) enthält die Anzeige `Integer.parseInt` und die Methode `main`.) Sowohl

`getMessage` als auch `printStackTrace` liefern Informationen, die Ihnen helfen sollen, die Quelle der Probleme zu finden, die das Programm hat.



```
<terminated> InventoryB [Java Application] C:\Program Files\Java\jre8\bin\javaw.exe (15.04.2014 18:59:05)
Wie viele Kartons haben wir? drei
Meldung: ***For input string: "drei"***
java.lang.NumberFormatException: For input string: "drei"
    at java.lang.NumberFormatException.forInputString(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at InventoryB.main(InventoryB.java:17)
```

Abbildung 13.4: Die Methoden eines Ausnahmeobjekts aufrufen



Wenn Sie Aufrufe von `System.out.println` mit Aufrufen von `printStackTrace` kombinieren, ist die Reihenfolge, in der Java die Informationen anzeigt, nicht vorhersehbar. So kann zum Beispiel in [Abbildung 13.4](#) der Text Nachricht: `***For input string: "drei"***` vor oder hinter dem *Stacktrace* (wie die reine Java-Fehlermeldung genannt wird) angezeigt werden. Wenn die Reihenfolge dieser Ausgabe wichtig ist, ändern Sie `out.println("Message: ***" in`
`System.err.println("Message: ***").`

Ausnahmetypen

Was kann heute sonst noch schiefgehen? Gibt es noch weitere Arten von Ausnahmen – Dinge, die nicht der Klasse `NumberFormatException` entstammen? Aber sicher doch, dort draußen warten noch viele Arten von Ausnahmen auf Sie. Sie können sogar eigene Ausnahmen erstellen. Sie wollen das ausprobieren? Schauen Sie sich die [Listings 13.3](#) und [13.4](#) an.

```
@SuppressWarnings("serial")
class OutOfRangeException extends Exception {
}
```

[Listing 13.3](#): Stellen Sie Ihre eigene Ausnahme her.

```
import static java.lang.System.out;
import java.util.Scanner;
import java.text.NumberFormat;
public class InventoryC {
    public static void main(String args[]) {
        final double boxPrice = 3.25;
        Scanner keyboard = new Scanner(System.in);
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();
        out.print("Wie viele Kartons haben wir? ");
        String numBoxesIn = keyboard.next();
        try {
            int numBoxes = Integer.parseInt(numBoxesIn);

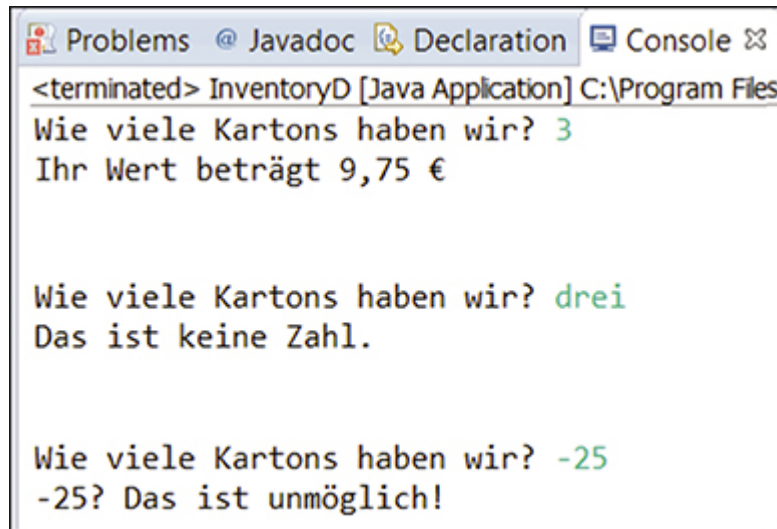
            if (numBoxes < 0) {
                throw new OutOfRangeException();
            }

            out.print("Ihr Wert beträgt ");
            out.println(
                currency.format(numBoxes * boxPrice));
        } catch (NumberFormatException e) {
            out.println("Das ist keine Zahl.");
        } catch (OutOfRangeException e) {
            out.print(numBoxesIn);
            out.println("? Das ist unmöglich!");
        }
        keyboard.close();
    }
}
```

[Listing 13.4](#): Die benutzerdefinierte Ausnahme verwenden

Die Codes in den [Listings 13.3](#) und [13.4](#) beheben ein Problem, das in [Abbildung 13.3](#) auftaucht. Der Benutzer notiert, dass im Regal –25 Kartons liegen, und der Computer nimmt diesen Wert, ohne zu zögern, als gegeben an. Die Wahrheit ist, dass Sie ein Schwarzes Loch (oder ein anderes außergewöhnliches Phänomen des Star-Trek-Zeitalters) benötigen, um in irgendeinem Regal eine negative Anzahl von Kartons zu lagern. Deshalb sollte das Programm angehalten werden, wenn ein

Benutzer negative Zahlen eingibt – und das ist genau das, was der Code in [Listing 13.4](#) bewirkt. [Abbildung 13.5](#) zeigt die entsprechende Reaktion des Computers.



```
<terminated> InventoryD [Java Application] C:\Program Files
Wie viele Kartons haben wir? 3
Ihr Wert beträgt 9,75 €

Wie viele Kartons haben wir? drei
Das ist keine Zahl.

Wie viele Kartons haben wir? -25
-25? Das ist unmöglich!
```

[Abbildung 13.5](#): Der Code aus den [Listings 13.3](#) und [13.4](#) wird dreimal ausgeführt.

Der Code in [Listing 13.3](#) deklariert eine neue Ausnahmeklasse – `OutOfRangeException`. Häufig macht es Sinn, eine negative Zahl einzugeben, weshalb `OutOfRangeException` kein Bestandteil der Java-API ist. Natürlich sollten negative Zahlen im Inventurprogramm als Anomalie gekennzeichnet werden.

In [Listing 13.3](#) gewinnt die Klasse `OutOfRangeException` den Preis für das kleinste abgeschlossene Stückchen Code in diesem Buch. Der Code der Klasse besteht nur aus einer Deklarationszeile und einem leeren Paar Klammern. Nur der Ausdruck `extends Exception` ist für die Funktionsfähigkeit zuständig. Da es sich bei der Klasse `OutOfRangeException` um eine Unterklasse der Java-API-Klasse `Exception` handelt, kann jede ihrer Instanzen geworfen werden.

In [Listing 13.4](#) wird eine neue Instanz von `OutOfRangeException` geworfen. Wenn dies geschieht, fängt die Klausel `catch (OutOfRangeException e)` die Instanz. Die

Klausel gibt die Eingabe des Benutzers wieder und zeigt die Meldung `Das ist unmöglich!` an.

Der Text `@SuppressWarnings("serial")` in [Listing 13.3](#) ist eine Annotation. Mehr über Annotations finden Sie in [Kapitel 8](#). In [Kapitel 9](#) stehen ein paar Worte zur Annotation `SuppressWarnings`.



Wenn Sie Eclipse einsetzen, kann es sein, dass Sie in [Listing 13.4](#) neben der Zeile `new OutOfRangeException()` einen gelben Warnhinweis zu sehen bekommen. Wenn Sie den Mauszeiger über diesem Warnhinweis schweben lassen, teilt Ihnen Eclipse mit: »Resource leak: 'keyboard' is not closed at this location.« Damit weist die IDE noch einmal ausdrücklich darauf hin, dass `keyboard` nicht geschlossen worden ist. Eclipse ist wirklich kleinlich, wenn es darum geht, dafür zu sorgen, dass Ihr Code gegebenenfalls den Befehl `keyboard.close()` auch ausführen kann. Aber meiner Meinung nach können Sie diese Warnung gefahrlos ignorieren.

Wer fängt die Ausnahme ein?

Werfen Sie noch einmal einen Blick auf [Listing 13.4](#). Beachten Sie, dass eine einzelne `try`-Klausel von mehreren `catch`-Klauseln begleitet werden kann. Wenn eine Ausnahme innerhalb einer `try`-Klausel geworfen wird, beginnt der Computer damit, die begleitende Liste mit den `catch`-Klauseln abzuarbeiten. Dabei beginnt er mit der `catch`-Klausel, die der `try`-Klausel unmittelbar folgt, und arbeitet dann den Text des Programms ab.

Bei jeder `catch`-Klausel fragt sich der Computer selbst: »Ist die Ausnahme, die gerade geworfen worden ist, eine Instanz der Klasse, die Bestandteil der Parameterliste der aktuellen Klausel ist?«

- ✓ Falls nein, übergeht der Computer diese `catch`-Klausel und macht mit der nächsten `catch`-Klausel weiter.

- ✓ Falls ja, führt der Computer die `catch`-Klausel aus und überspringt alle anderen `catch`-Klauseln, die zur `try`-Klausel gehören. Der Computer macht weiter und führt die Befehle aus, die dem gesamten `try-catch`-Befehl folgen.

Für konkrete Beispiele zu diesem Thema siehe die [Listings 13.5](#) und [13.6](#).

```
@SuppressWarnings("serial")
class NumberTooLargeException
    extends OutOfRangeException {
}
```

[Listing 13.5](#): Noch ein Beispiel

```
import static java.lang.System.out;
import java.util.Scanner;
import java.text.NumberFormat;
public class InventoryD {
    public static void main(String args[]) {
        final double boxPrice = 3.25;
        Scanner keyboard = new Scanner(System.in);
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();
        out.print("Wie viele Kartons haben wir? ");
        String numBoxesIn = keyboard.next();
        try {
            int numBoxes = Integer.parseInt(numBoxesIn
);
                if (numBoxes < 0) {
                    throw new OutOfRangeException();
                }
                if (numBoxes > 1000) {
                    throw new NumberTooLargeException();
                }

                out.print("Ihr Wert beträgt ");
                out.println(
                    currency.format(numBoxes * boxPrice));
            }

            catch (NumberFormatException e) {
                out.println("Das ist keine Zahl.");
            }
        }
    }
}
```

```

        catch (OutOfRangeException e) {
            out.print(numBoxesIn);
            out.println("? Das ist unmöglich!");
        }
        catch (Exception e) {
            out.print("Irgendetwas ist nicht in Ordnung
, ");
            out.print("und ich weiß nicht, ");
            out.println("was die Ursache ist.");
        }
        out.println("Damit ist die Sache erledigt.");
        keyboard.close();
    }
}

```

[Listing 13.6](#): Wer bekommt den Schwarzen Peter?

Damit Sie den Code aus den [Listings 13.5](#) und [13.6](#) ausführen können, benötigen Sie eine weitere Java-Programmdatei, die Klasse `OutOfRangeException` aus [Listing 13.3](#).

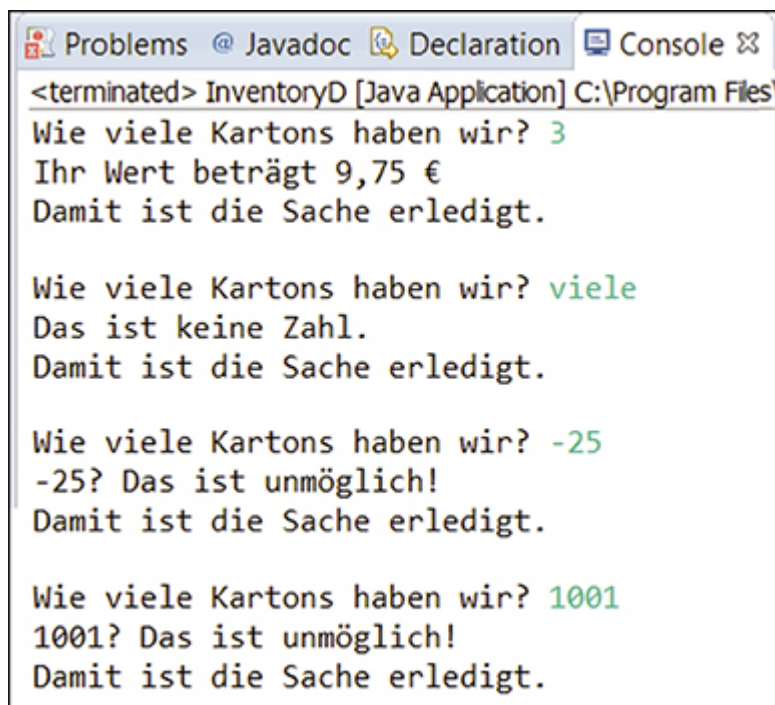
[Listing 13.6](#) spricht ein Szenario an, bei dem im Lager nur begrenzt Platz vorhanden ist. Sie können maximal 1.000 Kartons unterbringen, und ab und zu fragt das Programm nach, wie viele Kartons es im Lager gibt. Wenn dann jemand irrtümlich 100000 eingibt, überprüft das Programm die Lage und verwirft alles, was größer als 1.000 ist.

[Listing 13.6](#) achtet auf eine `NumberTooLargeException`, aber damit das Leben nicht zu langweilig wird, enthält [Listing 13.6](#) für diese Ausnahme keine `catch`-Klausel.

Da `NumberTooLargeException` eine Unterklasse von `OutOfRangeException` ist, handelt es sich bei `NumberTooLargeException` nur um eine besondere Form von `OutOfRangeException`. Also beginnt der Computer in [Listing 13.6](#) damit, dass er eine Klausel sucht, um eine `NumberTooLargeException` zu fangen. Wenn der Computer über die `catch`-Klausel `OutOfRangeException` stolpert, hat er etwas Passendes gefunden und führt die Befehle in dieser `catch`-Klausel aus.

Um diese Geschichte nicht immer wieder aufs Neue erzählen zu müssen, führe ich eine neue Terminologie ein. Ich sage, dass die `catch`-Klausel `OutOfRangeException` mit Parameter zur `NumberTooLargeException` *passt*, die geworfen wird. Ich nenne eine solche `catch`-Klausel eine *passende catch-Klausel*.

Die folgende Auflistung beschreibt verschiedene Dinge, die ein Benutzer machen könnte, und wie der Computer darauf reagiert. Wenn Sie sich diese Liste vornehmen, sollten Sie immer wieder einen Blick auf [Abbildung 13.6](#) werfen, die vier Ausführungen des Programms zeigt.



```
Problems @ Javadoc Declaration Console x
<terminated> InventoryD [Java Application] C:\Program Files
Wie viele Kartons haben wir? 3
Ihr Wert beträgt 9,75 €
Damit ist die Sache erledigt.

Wie viele Kartons haben wir? viele
Das ist keine Zahl.
Damit ist die Sache erledigt.

Wie viele Kartons haben wir? -25
-25? Das ist unmöglich!
Damit ist die Sache erledigt.

Wie viele Kartons haben wir? 1001
1001? Das ist unmöglich!
Damit ist die Sache erledigt.
```

[Abbildung 13.6](#): Der Code in [Listing 13.6](#) wurde viermal ausgeführt.

- ✓ **Der Benutzer gibt eine normale Ganzzahl wie die Zahl 3 ein.** Alle Befehle in der `try`-Klausel werden ausgeführt. Dann lässt der Computer alle `catch`-Klauseln aus und beschäftigt sich mit dem Code, der unmittelbar auf die letzte `catch`-Klausel folgt (siehe [Abbildung 13.7](#)).

✓ **Der Benutzer gibt etwas ein, das keine Ganzzahl ist (zum Beispiel das Wort *vieler*).** Der Computer wirft eine `NumberFormatException` und überspringt die restlichen Befehle in der `try`-Klausel. Der Computer führt die Befehle in der ersten `catch`-Klausel aus – der Klausel, deren Parameter vom Typ `NumberFormatException` ist. Dann überspringt der Computer die restlichen `catch`-Klauseln und führt den Code aus, der unmittelbar auf die `catch`-Klauseln folgt (siehe [Abbildung 13.8](#)).

```
try {  
    // Normale Verarbeitung (keine Ausnahme werfen)  
  
    catch (NumberFormatException e) {  
        out.println("Das ist keine Zahl.");  
    }  
  
    catch (OutOfRangeException e) {  
        out.print(numBoxesIn);  
        out.println("? Das ist unmöglich!");  
    }  
  
    catch (Exception e) {  
        out.print("Irgendetwas ist nicht ")  
        out.print("in Ordnung, ");  
        out.print("und ich weiß nicht, ");  
        out.println("was die Ursache ist.");  
    }  
    out.println("Damit ist die Sache erledigt.");  
}
```

[Abbildung 13.7](#): Es wird keine Ausnahme geworfen.

```

try {
    // Eine neue NumberFormatException werfen
    catch (NumberFormatException e) {
        out.println("Das ist keine Zahl.");
    }

    catch (OutOfRangeException e) {
        out.print(numBoxesIn);
        out.println("? Das ist unmöglich!");
    }

    catch (Exception e) {
        out.print("Irgendetwas ist nicht ")
        out.print("in Ordnung, ");
        out.print("und ich weiß nicht, ");
        out.println("was die Ursache ist.");
    }

    out.println("Damit ist die Sache erledigt.");
}

```

Abbildung 13.8: Eine *NumberFormatException* wird geworfen.

- ✓ **Der Benutzer gibt eine negative Zahl wie -25 ein.** Der Code wirft eine *OutOfRangeException* Exception. Der Computer übergeht die restlichen Befehle in der *try*-Klausel. Er übergeht sogar die Befehle in der ersten *catch*-Klausel. (Schließlich ist eine *OutOfRangeException* Exception keine Form der *NumberFormatException*. Die *catch*-Klausel mit dem Parameter *NumberFormatException* passt nicht zu dieser *OutOfRangeException*.) Der Computer führt die Befehle in der zweiten *catch*-Klausel aus – der Klausel, deren Parameter vom Typ *OutOfRangeException* ist. Dann übergeht der Computer die dritte *catch*-Klausel und führt den Code aus, der unmittelbar auf die *catch*-Klauseln folgt (siehe [Abbildung 13.9](#)).

```

try {
    // Eine neue OutOfRangeException werfen

    catch (NumberFormatException e) {
        out.println("Das ist keine Zahl.");
    }

    catch (OutOfRangeException e) {
        out.print(numBoxesIn);
        out.println("? Das ist unmöglich!");
    }

    catch (Exception e) {
        out.print("Irgendetwas ist nicht ")
        out.print("in Ordnung, ");
        out.print("und ich weiß nicht, ");
        out.println("was die Ursache ist.");
    }

    out.println("Damit ist die Sache erledigt.");
}

```

Abbildung 13.9: Es wird eine *OutOfRangeException* geworfen.

✓ **Der Benutzer gibt einen unrealistisch großen Wert wie die Zahl 1001 ein.** Der Code wirft eine *NumberTooLargeException*. Der Computer überspringt die restlichen Befehle in der *try*-Klausel und die Befehle in der ersten *catch*-Klausel. (Schließlich ist eine *NumberTooLargeException* keine Form der *NumberFormatException*.)

Aber die *NumberTooLargeException* ist, wie es der Code in [Listing 13.5](#) beschreibt, eine Unterklasse der Klasse *OutOfRangeException*. Wenn der Computer die *catch*-Klausel erreicht, stellt er fest, dass die

`NumberTooLargeException` eine Art von `OutOfRangeException` ist. Er führt die Befehle in der `catch`-Klausel aus – der Klausel, die einen Parameter vom Typ `OutOfRangeException` enthält. Beide Ausnahmen passen also zusammen.

Der Computer führt nun die Befehle innerhalb der zweiten `catch`-Klausel aus. Dann überspringt er die dritte `catch`-Klausel und führt den Code aus, der unmittelbar auf alle `catch`-Klauseln folgt (siehe [Abbildung 13.10](#)).

```
try {  
    // Eine NumberTooLargeException werfen  
    catch (NumberFormatException e) {  
        out.println("Das ist keine Zahl.");  
    }  
    catch (OutOfRangeException e) {  
        out.print(numBoxesIn);  
        out.println("? Das ist unmöglich!");  
    }  
    catch (Exception e) {  
        out.print("Irgendetwas ist nicht ")  
        out.print("in Ordnung, ");  
        out.print("und ich weiß nicht, ");  
        out.println("was die Ursache ist.");  
    }  
    out.println("Damit ist die Sache erledigt.");  
}
```

[Abbildung 13.10](#): Eine `NumberTooLargeException` ausführen

✓ **Es geschieht etwas Unvorhersehbares. (Ich weiß nicht, was.)** Bei meinem unstillbaren Drang herumzuspielen, habe

ich mir die `try`-Klausel in [Listing 13.6](#) genommen und einen Befehl hinzugefügt, der eine `IOException` wirft. Dafür gibt es eigentlich keinen Grund – ich wollte einfach sehen, was dann passiert.

Als der Code eine `IOException` warf, übersprang der Computer die restlichen Befehle in der `try`-Klausel und die ersten beiden `catch`-Klauseln. Wenn der Computer die dritte `catch`-Klausel erreicht, ist er der Meinung, dass eine `IOException` eine Art von `Exception` ist. Er findet also eine passende `catch`-Klausel – eine Klausel mit einem Parameter vom Typ `Exception` – und führt die darin enthaltenen Befehle aus. Danach macht er mit dem Code weiter, der allen `catch`-Klauseln unmittelbar folgt (siehe [Abbildung 13.11](#)).

Wenn der Computer nach passenden `catch`-Klauseln sucht, klinkt er sich in die oberste Klausel ein, die der folgenden Beschreibung entspricht:

- ✓ Der Parametertyp der Klausel ist mit dem Typ der Ausnahme identisch, die geworfen worden ist.
- ✓ Der Parametertyp der Klausel ist eine Superklasse des Typs der Ausnahme.

Wenn es weiter unten in der Liste der `catch`-Klauseln eine bessere Übereinstimmung geben sollte, haben Sie Pech gehabt. Stellen Sie sich vor, dass Sie dem Code in [Listing 13.6](#) eine `catch`-Klausel mit einem Parameter vom Typ `NumberTooLargeException` hinzugefügt haben. Stellen Sie sich weiter vor, dass Sie die neue `catch`-Klausel *hinter* der Klausel mit dem Parametertyp `OutOfRangeException` platziert haben. Da die `NumberTooLargeException` eine Unterklasse der Klasse `OutOfRangeException` ist, wird der Code in Ihrer neuen `NumberTooLargeException`-Klausel niemals ausgeführt. So ist nun mal das Leben!


```

try {
    // Eine neue IOException werfen

    catch (NumberFormatException e) {
        out.println("Das ist keine Zahl.");
    }

    catch (OutOfRangeException e) {
        out.print(numBoxesIn);
        out.println("? Das ist unmöglich!");
    }

    catch (Exception e) {
        out.print("Irgendetwas ist nicht ")
        out.print("in Ordnung, ");
        out.print("und ich weiß nicht, ");
        out.println("was die Ursache ist.");
    }

    out.println("Damit ist die Sache erledigt.");
}

```

Abbildung 13.11: Es wird eine *IOException* geworfen.

Die Multi-catch-Klausel

Seit Java 7 können Sie in einer einfachen `catch`-Klausel mehr als eine Art von Ausnahmen fangen. So möchten Sie zum Beispiel in einem Inventurprogramm nicht zwischen dem Werfen einer `NumberFormatException` und Ihrer eigenen `OutOfRangeException` unterscheiden. In diesem Fall können Sie einen Teil von [Listing 13.6](#) wie folgt neu schreiben:

```

try {
    int numBoxes = Integer.parseInt(numBoxesIn);
    if (numBoxes < 0) {
        throw new OutOfRangeException();
    }
    if (numBoxes > 1000) {
        throw new NumberTooLargeException();
    }
    out.print("Ihr Wert beträgt ");
    out.println(
        currency.format(numBoxes * boxPrice));
}

```

```

}
catch (NumberFormatException | OutOfRangeException e)
{
    out.print(numBoxesIn);
    out.println("? Das ist unmöglich!");
}
catch (Exception e) {
    out.print("Irgendetwas ist nicht in Ordnung, ");
    out.print("und ich weiß nicht, ");
    out.println("was die Ursache ist.");
}

```

Das Pipe-Symbol (|) weist Java an, entweder eine `NumberFormatException` oder eine `OutOfRangeException` zu fangen. Wenn Sie eine Ausnahme werfen, die von einem dieser beiden Typen ist, zeigt das Programm den Wert von `numBoxesIn` an, dem der Text `? Das ist unmöglich!` folgt. Wenn Sie eine Ausnahme werfen, die weder eine `NumberFormatException` noch eine `OutOfRangeException` ist, springt das Programm zur letzten `catch`-Klausel und zeigt `Irgendetwas ist nicht in Ordnung, und ich weiß nicht ... an`.

Alle Bedenken in den Wind schlagen

Gehören Sie zu den Zwangsneurotikern? Möchten Sie jede nur denkbare Ausnahme fangen, bevor sie möglicherweise Ihr Programm zum Absturz bringt? Dann aufgepasst!. Java verhindert, dass Sie paranoid werden. Sie können keine Ausnahme fangen, die keine Chance hat, geworfen zu werden.

Schauen Sie sich den folgenden Code an. Er enthält in einer `try`-Klausel eine völlig unschuldige Anweisung `i++`. Schön und gut. Aber dann täuscht die `catch`-Klausel des Codes vor, eine `IOException` zu fangen.

```

// Schlechter Code!
try {
    i++;
} catch (IOException e) {
    e.printStackTrace();
}

```

Wen versucht diese `catch`-Klausel zu beeindrucken? Eine Anweisung wie `i++` sorgt weder für eine Eingabe noch für eine Ausgabe. Der Code in der `try`-Klausel ist möglicherweise noch nicht einmal in der Lage, eine `IOException` zu werfen. Also meldet sich der Compiler bei Ihnen und sagt: »Hallo, `catch`-Klausel. Wach auf. Komm von deinem hohen Ross herunter.« Aber wollen wir genauer sein. Der Rüffel des Compilers lautet in Wirklichkeit:

```
exception java.io.IOException is never thrown in body  
of corresponding try statement
```

Sie werden damit darüber informiert, dass im Körper der entsprechenden `catch`-Klausel niemals eine `IOException` geworfen wird.

Etwas Sinnvolles tun

Bisher hat noch jedes Beispiel in diesem Kapitel eine Ausnahme gefangen, eine Nachricht wie `Falsche Eingabe` ausgegeben und sich dann sauber beendet. Wäre es da nicht ganz nett, einmal ein Programm zu Gesicht zu bekommen, das weitermacht, nachdem eine Ausnahme gefangen wurde? [Listing 13.7](#) enthält in einer Schleife einen `try-catch`-Befehl. Die Schleife läuft so lange, bis der Benutzer etwas Sinnvolles eingibt.

```
import static java.lang.System.out;
import java.util.Scanner;
import java.text.NumberFormat;
public class InventoryLoop {
    public static void main(String args[]) {
        final double boxPrice = 3.25;
        boolean gotGoodInput = false;
        Scanner keyboard = new Scanner(System.in);
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();
        do {
            out.print("Wie viele Kartons haben wir? ");
            String numBoxesIn = keyboard.next();
            try {
                int numBoxes = Integer.parseInt(numBoxesIn);
            } catch (NumberFormatException e) {
                out.print("Das ist keine Zahl ");
            }
        } while (!gotGoodInput);
    }
}
```

```

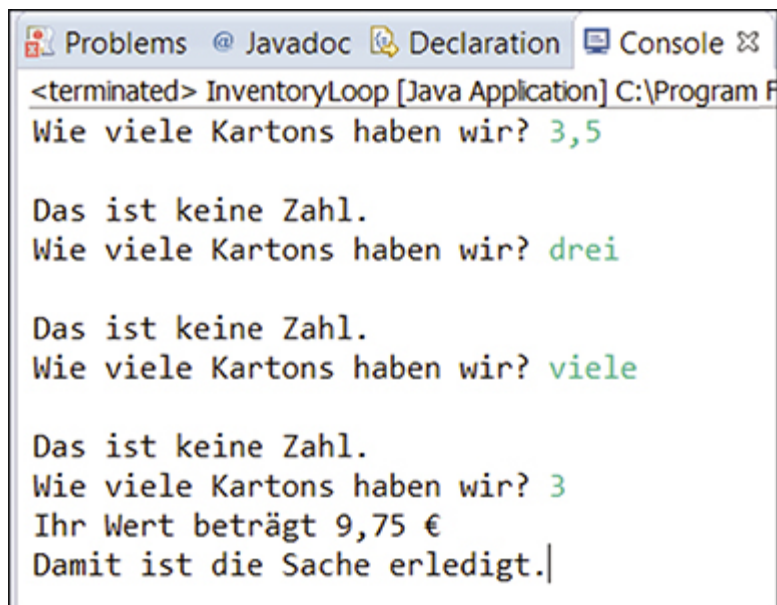
        out.println
            (currency.format(numBoxes * boxPrice))
;

        gotGoodInput = true;
    } catch (NumberFormatException e) {
        out.println();
        out.println("Das ist keine Zahl.");
    }
} while (!gotGoodInput);
out.println("Damit ist die Sache erledigt.");
keyboard.close();
}
}

```

[Listing 13.7](#): Weitermachen!

[Abbildung 13.12](#) zeigt, wie es aussieht, wenn der Code aus [Listing 13.7](#) ausgeführt wird. Bei den ersten drei Versuchen gibt der Benutzer alles, aber keine gültige Zahl ein. Erst der vierte Versuch ist erfolgreich. Der Benutzer gibt 3 ein, und der Computer verlässt die Schleife.



```

<terminated> InventoryLoop [Java Application] C:\Program F
Wie viele Kartons haben wir? 3,5

Das ist keine Zahl.
Wie viele Kartons haben wir? drei

Das ist keine Zahl.
Wie viele Kartons haben wir? viele

Das ist keine Zahl.
Wie viele Kartons haben wir? 3
Ihr Wert beträgt 9,75 €
Damit ist die Sache erledigt.

```

[Abbildung 13.12](#): Der Code aus [Listing 13.7](#) wird ausgeführt.

Unsere Freunde, die guten Ausnahmen

Es geht das Gerücht, dass Java-Ausnahmen ihren Ursprung immer in unerwünschten Fehlersituationen haben. Obwohl viel Wahres an diesem Gerücht ist, ist es nicht ganz richtig. Gelegentlich kommt es auch zu geplanten Ausnahmen. Nehmen Sie zum Beispiel die Entdeckung, dass eine Datei zu Ende ist. Der folgende Code kopiert eine Datei:

```
try {
    while (true) {
        dataOut.writeByte(dataIn.readByte());
    }
} catch (EOFException e) {
    numFilesCopied = 1;
}
```

Um Bytes von `dataIn` nach `dataOut` zu kopieren, begeben Sie sich einfach in eine `while`-Schleife. Diese Schleife scheint aufgrund ihrer `true`-Bedingung unendlich zu laufen. Aber irgendwann erreichen Sie das Ende der `dataIn`-Datei. (Ein solches Ende wird als *EOF* – für *End of File* oder *Ende der Datei* – bezeichnet.) In diesem Fall wirft die Methode `readByte` eine `EOFException` (eine *End-of-File*-Ausnahme). Das Werfen dieser Ausnahme sorgt dafür, dass der Computer die `try`-Klausel und die `while`-Schleife verlässt. Danach können Sie in der `catch`-Klausel das machen, was Sie wollen, und die normale Verarbeitung fortsetzen.

Eine Ausnahme verarbeiten oder den Schwarzen Peter weitergeben

Sie glauben also, dass Sie Java kennen? Sie sagen, dass Sie es schon bis [Kapitel 13](#) geschafft haben? Ich bin beeindruckt. Sie müssen richtig hart gearbeitet haben.

Was halten Sie davon, einmal eine Pause einzulegen? Ein kurzes Nickerchen kann Wunder wirken. Was halten Sie von zehn Sekunden? Oder ist das zu lang? Machen Sie fünf Sekunden daraus.

[Listing 13.8](#) enthält ein Programm, das seine Ausführung fünf Sekunden lang unterbrechen soll. Das Problem dabei ist, dass das Programm in [Listing 13.8](#) nicht ganz richtig ist. Schauen Sie sich das Listing eine Minute lang an, und dann erkläre ich Ihnen, was am Programm falsch ist.

```
/*
 * Dieser Code wird nicht kompiliert.
 */
import static java.lang.System.out;
public class NoSleepForTheWeary {
    public static void main(String args[]) {
        out.print("Entschuldigen Sie mich bitte - ");
        out.println("fünf Sekunden Pause...");
        takeANap();
        out.println("Ah, das war erfrischend.");
    }
    static void takeANap() {
        Thread.sleep(5000);
    }
}
```

[Listing 13.8](#): Ein fehlerhaftes Programm

Die Strategie in [Listing 13.8](#) ist nicht übel. Der Plan sieht so aus, dass die Methode `sleep` aufgerufen wird, die Bestandteil der Java-API ist. Die Methode `sleep` gehört zur API-Klasse `Thread`. Wenn Sie diese Methode aufrufen, repräsentiert die Zahl, die Sie als Dauer der Pause eingeben, Millisekunden.

`Thread.sleep(5000)` bedeutet somit, dass das Programm fünf Sekunden lang pausiert.

Das Problem besteht darin, dass der Code in der Methode `sleep` eine Ausnahme werfen kann. Bei dieser Art von Ausnahme handelt es sich um eine Instanz der Klasse `InterruptedException`. Wenn Sie versuchen, den Code in [Listing 13.8](#) zu kompilieren, erhalten Sie eine Meldung wie diese hier:

unreported exception java.lang.InterruptedException;
must be caught or declared to be thrown

Diese Meldung sollte eigentlich (lesbar gemacht und übersetzt) so aussehen:

Nicht verarbeitete Ausnahme vom Typ InterruptedException

Ob so oder so, auf jeden Fall gilt, dass diese Meldung nicht willkommen ist.



Damit Sie verstehen, was Ausnahmen im Allgemeinen sind, müssen Sie nicht genau wissen, was eine `InterruptedException` ist. Es reicht, wenn Sie wissen, dass der Aufruf von `Thread.sleep` eines dieser `InterruptedException`-Objekte wirft. Wenn Sie aber wirklich daran interessiert sind: Eine `InterruptedException` wird geworfen, wenn Code den Schlaf eines anderen Codes unterbricht (*to interrupt* bedeutet auf Deutsch *unterbrechen*). Stellen Sie sich vor, dass Sie zwei Codestücke haben, die gleichzeitig laufen. Eines dieser Codestücke ruft die Methode `Thread.sleep` auf. Wenn nun das zweite Codestück die Methode `interrupt` aufruft, sorgt es dafür, dass die Methode `Thread.sleep` eine Vollbremsung hinlegen muss. Die Methode `Thread.sleep` reagiert, indem sie eine `InterruptedException` wirft.

Die Programmiersprache Java kennt zwei verschiedene Arten von Ausnahmen. Sie werden *geprüfte* und *ungeprüfte* Ausnahmen genannt:

- ✓ Dass möglicherweise eine geprüfte Ausnahme geworfen wird, muss im Code bestätigt werden.
- ✓ Dass möglicherweise eine ungeprüfte Ausnahme geworfen wird, muss im Code nicht bestätigt werden.

Eine `InterruptedException` ist einer von Javas geprüften Ausnahmetypen. Wenn Sie eine Methode aufrufen, die das Potenzial besitzt, eine `InterruptedException` zu werfen, müssen Sie diese Ausnahme im Code bestätigen.

Was meine ich wohl damit, dass *eine Ausnahme im Code bestätigt wird*?

```
// Der Autor möchte der InterruptedException danken,  
// ohne die dieser Code nicht hätte geschrieben werde  
n  
// können.
```

Das ist es aber nicht, was *im Code bestätigt* bedeutet. Mit dem Bestätigen einer Ausnahme im Code ist eines dieser beiden Dinge gemeint:

- ✓ Die Befehle (einschließlich ihrer Methodenaufrufe), die die Ausnahme werfen können, befinden sich innerhalb einer `try`-Klausel. Diese `try`-Klausel verfügt über eine `catch`-Klausel mit einem passenden Ausnahmetyp in ihrer Parameterliste.
- ✓ Die Befehle (einschließlich ihrer Methodenaufrufe), die die Ausnahme werfen können, befinden sich innerhalb einer Methode, die in ihrem Kopf über eine `throws`-Klausel verfügt. Diese Klausel enthält einen passenden Ausnahmetyp.

Wenn Sie die Wortwahl in diesen beiden Punkten irritieren sollte, machen Sie sich nichts daraus. Die nächsten beiden Listings bringen Licht ins Dunkel.

In [Listing 13.9](#) befindet sich der Methodenaufruf, der eine `InterruptedException` werfen kann, in einer `try`-Klausel. Diese Klausel verfügt über eine `catch`-Klausel mit dem Ausnahmetyp `InterruptedException`.

```
import static java.lang.System.out;  
public class GoodNightsSleepA {  
    public static void main(String args[]) {  
        out.print("Entschuldigen Sie mich bitte - ");  
        out.println("fünf Sekunden Pause...");  
        takeANap();  
        out.println("Ah, das war erfrischend.");  
    }  
    static void takeANap() {  
        try {  
            Thread.sleep(5000);  
        } catch (InterruptedException e) {  
            out.println("Hallo, wer hat mich aufgeweckt  
?");  
        }  
    }  
}
```



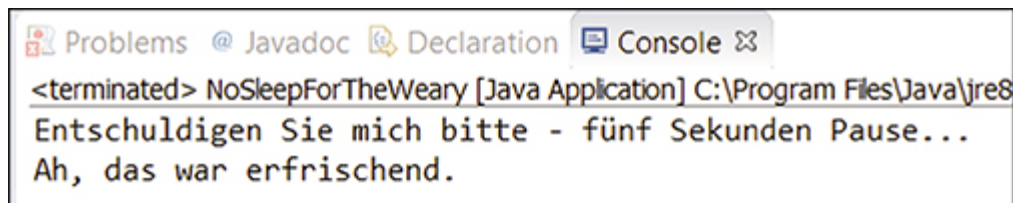
```

    }
}

```

[Listing 13.9](#): Bestätigung mit einem `try-catch-Befehl`

Normalerweise ist es an dieser Stelle meine Aufgabe, Sie darüber zu informieren, dass eine Ausführung von Listing *SoUndSo* in Abbildung XYZ gezeigt wird. Hier gibt es nun aber in [Abbildung 13.13](#) das Problem, dass es nicht nur darum geht, dem Code in [Listing 13.9](#) gerecht zu werden. Wenn Sie das Programm in diesem Listing ausführen, zeigt der Computer Entschuldigen Sie mich bitte - fünf Sekunden Pause an, unterbricht seine Arbeit für fünf Sekunden und gibt dann Ah, das war erfrischend aus. Der Code funktioniert, weil sich der Aufruf der Methode `sleep`, der eine `InterruptedException` werfen kann, innerhalb einer `try`-Klausel befindet. Diese Klausel enthält eine `catch`-Klausel, deren Ausnahme vom Typ `InterruptedException` ist.



[Abbildung 13.13](#): Vor der Zeile mit *Ah* gibt es eine Pause von fünf Sekunden.

So viel zum Bestätigen einer Ausnahme mit einem `try-catch`-Befehl. Es gibt aber noch einen anderen Weg, um eine Ausnahme zu bestätigen. [Listing 13.10](#) stellt diesen Weg vor.

```

import static java.lang.System.out;
public class GoodNightsSleepB {
    public static void main(String args[]) {
        out.print("Entschuldigen Sie mich bitte - ");
        out.println("fünf Sekunden Pause...");
        try {
            takeANap();
        } catch (InterruptedException e) {

```

```

        out.println("Hallo, wer hat mich aufgewec
kt?");
    }
    out.println("Ah, das war erfrischend.");
}
static void takeANap() throws InterruptedException
n {
    Thread.sleep(5000);
}
}

```

[Listing 13.10](#): Durch Werfen bestätigen

Wenn Sie sehen möchten, was eine Ausführung dieses Codes ausgibt, verweise ich auf [Abbildung 13.13](#). Denken Sie nur daran, dass der Computer vor der Ausgabe von *Ah, das war erfrischend* eine Pause von fünf Sekunden einlegt.

Der entscheidende Teil von [Listing 13.10](#) steht im Kopf der Methode `takeANap`. Dieser Kopf endet mit `throws InterruptedException`. Indem die Methode `takeANap` ankündigt, dass sie eine `InterruptedException` wirft, gibt sie den Schwarzen Peter weiter. Was die `throws`-Klausel wirklich sagt, ist: »Ich erkenne, dass eine Anweisung in dieser Methode die Möglichkeit besitzt, eine `InterruptedException` zu werfen, aber ich bestätige die Ausnahme nicht in einem `try-catch`-Befehl. Java-Compiler, sei mir deswegen nicht böse. Ich verfüge nicht über einen `try-catch`-Befehl. Ich übergebe die Verantwortung für die Bestätigung der Ausnahme an die Methode `main` (die Methode, die die Methode `takeANap` aufgerufen hat).«

Und wirklich, der Aufruf von `takeANap` befindet sich innerhalb einer `try`-Klausel. Diese Klausel verfügt über eine `catch`-Klausel mit einem Parametertyp `InterruptedException`. Damit ist alles in Ordnung. Die Methode `takeANap` übergibt die Verantwortung an die Methode `main`, die diese Verantwortung mithilfe eines entsprechenden `try-catch`-Befehls akzeptiert. Jeder ist glücklich, sogar der Java-Compiler.

Damit Sie die Klausel `throws` besser verstehen, sollten Sie sich ein Volleyballspiel vorstellen, bei dem der Volleyball eine

Ausnahme ist. Wenn ein Spieler des gegnerischen Teams serviert, wirft dieser Spieler die Ausnahme. Der Ball überquert das Netz und kommt zu Ihnen. Wenn Sie den Ball über das Netz zurückspielen, fangen Sie die Ausnahme. Aber wenn Sie den Ball an einen Mitspieler weiterleiten, verwenden Sie die Klausel `throws`. Sie sagen damit: »Hier, Mitspieler, das ist jetzt deine Ausnahme. Kümmere dich darum.«



Ein Befehl in einer Methode kann eine Ausnahme werfen, zu der es keine passende `catch`-Klausel gibt. Hierzu gehören auch Situationen, in denen sich der Befehl, der die Ausnahme wirft, noch nicht einmal im `try`-Block befindet. In diesem Fall verlässt das Programm die Methode, die den fehlerhaften Code enthält, und die Ausführung springt zu dem ersten Code zurück, der die Methode aufgerufen hat.



Eine Methode kann in ihrer `throws`-Klausel mehr als eine Ausnahme benennen. Verwenden Sie Kommata so, wie es das folgende Beispiel zeigt, um die Namen der Ausnahmetypen voneinander zu trennen:

```
throws InterruptedException, IOException,  
        ArithmeticException
```

Die Java-API kennt Hunderte von Ausnahmetypen. Einige davon sind Unterklassen der Klasse `RuntimeException`. Alles, was eine Unterklasse (oder eine Unter-Unterklasse oder eine Unter-Unter-Unterklasse und so weiter) ist, ist ungeprüft. Jede Ausnahme, die kein Abkömmling von `RuntimeException` ist, ist geprüft. Zu den ungeprüften Ausnahmen gehören Dinge, die ein Computer nur schwer vorhersehen kann, wie die `NumberFormatException` (unter anderem aus den [Listings 13.2](#) und [13.4](#)), die `ArithmeticException`, die `IndexOutOfBoundsException`, die berühmte `NullPointerException` und viele andere mehr. Wenn Sie Java-Code schreiben, ist dieser anfällig für Ausnahmen, aber das Einbinden dieses Codes in `try`-Klauseln (oder das Weitergeben

des Schwarzen Peters über `throws`-Klauseln) ist ausschließlich Ihre Aufgabe.

Außerdem enthält die Java-API auch ein gerüttelt Maß an geprüften Ausnahmen. Der Computer kann Ausnahmen dieser Art leicht entdecken. Aus diesem Grund besteht Java darauf, dass bei diesen Ausnahmen jeder Befehl, der möglicherweise Ausnahmen werfen könnte, mit einem `try`-Befehl oder einer `throws`-Klausel bestätigt wird. Zu den geprüften Java-Ausnahmen gehören die `InterruptedException` ([Listings 13.9](#) und [13.10](#)), die `IOException`, die `SQLException` und eine Menge anderer interessanter Ausnahmen.

Die Aufgabe mit der Klausel »finally« abschließen«

Es war einmal, da war ich ein junger Hüpfer, wohnte mit meinen Eltern in Philadelphia und hatte gerade meinen Führerschein gemacht. Ich war unterwegs zum Haus eines Freundes und war mit den Gedanken irgendwo, als plötzlich aus dem Nichts heraus ein Wagen auftauchte und in meine Beifahrertür krachte. So etwas nennt sich dann *BeiRoterAmpelFahrenException*.

Auf jeden Fall waren beide Autos noch fahrbereit, und wir befanden uns mitten auf einer Kreuzung. Um ein Verkehrschaos zu vermeiden, fuhren wir beide an den nächstgelegenen Straßenrand. Ich kramte nach meinem Führerschein und öffnete die Tür, um mein Fahrzeug zu verlassen.

Das war dann der Augenblick, als es zum zweiten Unfall kam. Als ich den Wagen verließ, kam ein Linienbus vorbei, erwischte mich und schleuderte mich mehrmals gegen mein Auto. So etwas nennt sich dann *DaFreutSichEinAnwaltException*.

Glücklicherweise endete alles gut. Ich war zwar verletzt, aber nicht ernsthaft. Meine Eltern bezahlten den Schaden am anderen Auto, weshalb ich keine finanziellen Konsequenzen zu tragen hatte. (Ich konnte die finanziellen Lasten umgehen, indem ich die

BeiRoterAmpelFahrenException in meiner throws-Klausel unterbrachte.)

Dieser Unfall hilft dabei zu erklären, warum ich auf eine bestimmte Weise über den Umgang mit Ausnahmen denke. Insbesondere frage ich mich: »Was geschieht, wenn es zu einer zweiten Ausnahme kommt, während der Computer noch mit einer ersten Ausnahme zu kämpfen hat?« Schließlich sind auch die Anweisungen in einer `catch`-Klausel nicht vor Schwierigkeiten gefeit.

Die Antwort auf diese Frage ist alles andere als einfach. Für den Anfang können Sie in einer `catch`-Klausel eine `try`-Anweisung unterbringen. Dies schützt Sie vor unerwarteten, möglicherweise sogar fatalen Vorfällen, zu denen es während der Ausführung der `catch`-Klausel kommen kann. Aber wenn Sie erst einmal anfangen, sich über kaskadierende Ausnahmen Gedanken zu machen, öffnen Sie eine Büchse der Pandora. Es gibt unendlich viele Szenarien, und alles kann sehr schnell sehr kompliziert werden.

Eine Sache, die nicht zu schwierig ist, können Sie auf jeden Fall erledigen: eine `finally`-Klausel erstellen. Eine solche Klausel steht – wie die `catch`-Klausel – hinter einer `try`-Klausel. Der Unterschied zur `catch`-Klausel besteht darin, dass die Befehle in der `finally`-Klausel unabhängig davon ausgeführt werden, ob eine Ausnahme geworfen wird oder nicht. (*Finally* bedeutet auf Deutsch *letztendlich, zum Schluss*.) [Listing 13.11](#) enthält ein Beispiel.

```
import static java.lang.System.out;
public class DemoFinally {
    public static void main(String args[]) {
        try {
            doSomething();
        } catch (Exception e) {
            out.println("Ausnahme in main gefangen.");
        }
    }
    static void doSomething() {
```

```

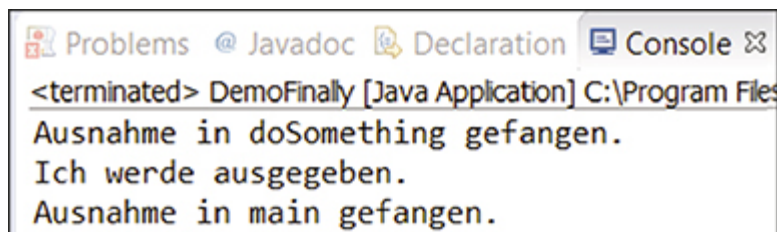
try {
    out.println(0 / 0);
} catch (Exception e) {
    out.println(
        "Ausnahme in doSomething gefangen.");
    out.println(0 / 0);
} finally {
    out.println("Ich werde ausgegeben.");
}
out.println("Ich werde nicht ausgegeben.");
}
}

```

[Listing 13.11](#): Hin und her springen

Wenn ich normalerweise über eine `try`-Anweisung nachdenke, vergesse ich auch den Computer nicht, der sich von einer unangenehmen Situation erholt. Diese Erholung erhält ihren Platz innerhalb einer `catch`-Klausel. Danach macht der Computer mit den Befehlen weiter, die im Anschluss an die `try`-Anweisung kommen. Wenn aber während der Ausführung der `catch`-Klausel irgendetwas schiefgeht, kann dieses Bild sehr schnell anders aussehen.

[Abbildung 13.14](#) stellt grafisch dar, was das Programm in [Listing 13.11](#) liefert. Als Erstes wird die Methode `doSomething` aufgerufen. Dann bereitet diese dumme Methode Schwierigkeiten. Sie dividiert durch 0, was in keiner Programmiersprache zulässig ist. Diese verrückte Aktion der Methode `doSomething` wirft eine `ArithmeticException`, die von der `catch`-Klausel des Befehl gefangen wird.



[Abbildung 13.14](#): Der Code in [Listing 13.11](#) wurde ausgeführt.

Die zwielichtige Methode `doSomething` dividiert in der `catch`-Klausel noch einmal 0 durch 0. Dieses Mal befindet sich der Befehl, der für diese Division zuständig ist, nicht in einer schützenden `try`-Klausel. Das geht in Ordnung, weil `ArithmeticException` ungeprüft ist. (Es handelt sich um eine der Unterklassen von `RuntimeException`. Außerdem muss `ArithmeticException` auch nicht in einer `try`- oder `throws`-Klausel bestätigt werden. Einzelheiten zu diesem Thema finden Sie im vorherigen Abschnitt.)

Ob geprüft oder ungeprüft, das Werfen einer weiteren `ArithmeticException` bringt die Steuerung dazu, aus der Methode `doSomething` herauszuspringen. Aber bevor der Computer diese Methode verlässt, führt er den letzten Willen des `try`-Befehls aus: die Anweisungen in der Klausel `finally`. Aus diesem Grund lesen Sie in [Abbildung 13.14](#) Ich werde ausgegeben.

Interessanterweise sehen Sie in [Abbildung 13.14](#) die Wörter Ich werde nicht ausgegeben nicht. Da die Ausführung der `catch`-Klausel eine eigene, nicht gefangene Ausnahme wirft, geht der Computer niemals weiter als bis zur `try-catch-finally`-Anweisung.

Der Computer begibt sich wieder zurück zur Methode `main`, und zwar an die Stelle, an der er diese Methode verlassen hat. Dort sorgt das Missgeschick mit der `ArithmeticException` der Methode `doSomething` dafür, dass die Programmausführung in eine `catch`-Klausel springt. Der Computer gibt Ausnahme in `main` gefangen aus, und dieser schreckliche Albtraum einer Programmausführung wird beendet.

Ein »try«-Befehl mit Ressourcen

Stellen Sie sich ein Programm vor, das Eingaben aus zwei verschiedenen Dateien oder von einem `Scanner` und aus einer Datei auf der Festplatte erhält. Um sicherzustellen, dass Sie danach auch alles wieder sauber bereinigen, bringen Sie in einer

finally-Klausel Aufrufe der Methode `close` unter (siehe [Listing 13.12](#)).

```
import java.io.File;
import java.io.IOException;
import java.util.Scanner;
public class Main {
    public static void main(String args[]) {
        Scanner scan1 = null;
        Scanner scan2 = null;
        try {
            scan1 = new Scanner(new File("File1.txt"))
);
            scan2 = new Scanner(new File("File2.txt"))
);
            // Etwas Sinnvolles tun
        } catch (IOException e) {
            // Oops!
        } finally {
            scan1.close();
            scan2.close();
            System.out.println("Fertig!");
        }
    }
}
```

[Listing 13.12](#): Zwei Dateien verwenden

In der Theorie führt der Computer `scan1.close()` und `scan2.close()` immer unabhängig davon aus, was während der Ausführung der `try`-Klausel schiefgeht. Aber wollen wir doch realistisch sein. Ein anderer Programmierer könnte den Code ändern und `scan1` mitten in der `try`-Klausel schließen:

```
try {
    scan1 = new Scanner(new File("File1.txt"));
    scan2 = new Scanner(new File("File2.txt"));
    // Etwas Sinnvolles tun, aber auch ...
    scan1.close();
    scan1 = null;
} catch (IOException e) {
    // Oops!
} finally {
    scan1.close();
    scan2.close();
}
```



```

        System.out.println("Fertig!");
    }

```

Jetzt sitzen Sie in einer echten Klemme. Der Wert von `scan1` ist in der Klausel `finally` `null`. Der Aufruf von `scan1.close()` schlägt fehl, was das Programm veranlasst, eine `NullPointerException` zu werfen und seine Ausführung zu beenden, bevor der Aufruf von `scan2.close()` erreicht wird. Im allerschlechtesten Fall wird `scan2` nicht geschlossen, was wiederum `File2.txt` sperrt, wodurch kein anderes Programm diese Datei nutzen kann.

Wenn ein Programm mehrere Ressourcen (viele Dateien, eine Datenbank und eine Datei oder was Ihnen sonst noch vorschwebt) verwendet, wird der Aufbau von `try`-Befehlen richtig kompliziert. Sie können `try`-Befehle innerhalb von `catch`-Klauseln und in allen möglichen verrückten Kombinationen anlegen. Glücklicherweise kennt Java einen sehr guten Weg, um das Problem zu lösen. Sie können ab Java 7 einen *try-mit-Ressourcen-Befehl* erstellen. [Listing 13.13](#) zeigt Ihnen, wie das geht.

```

import java.io.File;
import java.io.IOException;
import java.util.Scanner;
public class NewMain {
    public static void main(String args[]) {
        try (Scanner scan1 =
            new Scanner(new File("File1.txt"));
            Scanner scan2 =
            new Scanner(new File("File2.txt")))
        {
            // Etwas Sinnvolles tun
        } catch (IOException e) {
            // Oops!
        }
        System.out.println("Fertig!");
    }
}

```

[Listing 13.13](#): Dafür sorgen, dass Ressourcen geschlossen werden

In [Listing 13.13](#) stehen die Deklarationen von `scan1` und `scan2` in Klammern hinter dem Wort `try`. Diese in Klammern gesetzten Deklarationen weisen Java an, `scan1` und `scan2` automatisch zu schließen, wenn die Befehle in der `try`-Klausel ausgeführt worden sind. Sie können in den Klammern einer `try`-Anweisung mehrere Ressourcen deklarieren. In diesem Fall schließt Java alle Ressourcen automatisch, wenn die Befehle in der `try`-Klausel ausgeführt worden sind. Wenn Sie wollen, können Sie auch `catch`-Klauseln und eine `finally`-Klausel hinzufügen. Sie dürfen dabei auf alle möglichen Ressourcen zugreifen (Dateien, Datenbanken, Serververbindungen und so weiter), und Sie können in aller Gemütsruhe zusehen, wie Java die Verbindungen automatisch bedient.

Ist das Leben nicht schön?

14

Programmteile gemeinsam nutzen lassen

In diesem Kapitel ...

- ▶ Namen vor anderen Klassen verbergen
 - ▶ Namen für andere Klassen offenlegen
 - ▶ Das Programm so einstellen, dass es den richtigen Mittelweg findet
-

Eine Unterhaltung über private Felder und Methoden (die auch Thema dieses Kapitels sind) ...

Ich esse mit befreundeten Arbeitskollegen zu Mittag. »Sie können deine E-Mails lesen«, meint einer meiner Freunde. Ein anderer stimmte dem zu: »Sie kennen jede einzelne Website, die du besuchst. Sie wissen, welche Produkte du kaufst, wie dein Abendessen aussieht, was du anziehst und was du denkst. Sie kennen sogar deine tiefsten und dunkelsten Geheimnisse. Ich wäre überhaupt nicht überrascht, wenn Sie wüssten, wann du sterben wirst.«

Eine dritte Stimme ertönt: »Wir sind an dem Punkt angelangt, an dem du dir nicht einmal mehr die Nase putzen kannst, ohne dass das jemand aufzeichnet. Vor ein paar Wochen habe ich eine Website besucht, und die Seite wünschte mir alles Gute zum Geburtstag. Woher wussten die, dass ich das war, und wieso haben die an meinen Geburtstag gedacht?«

»Jungs«, sagte die erste Person, »ich habe eine Erkennungsmarke an meinem Auto, mit der ich alle Mautstellen passieren kann. Sie sendet die Mautstelle, die ich gerade passiere, und belastet automatisch meine Kreditkarte. Einmal im

Monat erhalte ich eine Abrechnung, die mir zeigt, wo ich gewesen bin und wann das war. Ich bin erstaunt, dass auf der Liste nicht steht, wen ich besucht und was ich dort getan habe.«

Ich denke nach und sage mir: »Das ist alles Quatsch. Ich persönlich fühle mich geschmeichelt, wenn mein Arbeitgeber, die Regierung oder ein großes Unternehmen so sehr an mich denkt, dass sie meine ganzen Bewegungen aufzeichnen. Mir fällt es normalerweise schon schwer genug, die Aufmerksamkeit von Leuten auf mich zu lenken, wenn ich das möchte. Und die meisten Stellen, die meine Einkaufs- und Lesegewohnheiten protokollieren, sind noch nicht einmal in der Lage, meinen Namen richtig zu schreiben, wenn sie mir Junk-Mails senden. ›Dies ist eine persönliche Nachricht für Mr. Larry Burg. Ist Mr. Burg zu Hause?‹ Leute ausspähen ist so was von langweilig. Ich kann die Schlagzeile auf der Titelseite von *The Times* direkt vor mir sehen: ›Der Autor von *Java für Dummies* trägt sein Unterhemd falsch herum!‹ Toll!«

So denke ich für ein paar Sekunden und sage dann: »Sie sind hinter uns her. Kameras! Das ist der nächste Schritt – überall Kameras.«

Zugriffsmodifizierer

Wenn Sie *Java für Dummies* bis hierher gelesen haben, ist Ihnen vielleicht eines klar geworden: Die objektorientierte Programmierung ist ganz groß im Verbergen von Einzelheiten. Programmierer, die Code schreiben, sollen auf keinen Fall am Code anderer Programmierer herumbasteln. Das hat nicht nur etwas mit Sicherheit und Geheimhaltung zu tun, sondern auch mit Modularität. Wenn Sie Einzelheiten verbergen, verhindert dies, dass anderer Code an Ihrem komplexen Programm herumschraubt. Ihr Code liegt in eigenständigen, verwaltbaren Einheiten vor. Sie beschränken Komplexität auf ein Minimum. Sie machen weniger Fehler. Sie sparen Geld. Sie helfen dabei, den Weltfrieden aufrechtzuerhalten.

In anderen Kapiteln gibt es viele Beispiele, die zeigen, wie private Felder verwendet werden. Wenn ein Feld privat (`private`) deklariert wird, ist es für alles verborgen, was von außen kommt. Dieses Verbergen erweitert unter anderem die Modularität und minimiert die Komplexität.

Irgendwo in den Seiten dieses Buches verstecken sich auch Beispiele von Dingen, die öffentlich (`public`) deklariert werden. Von einem Feld, das öffentlich deklariert wird, ist alles so bekannt, wie das bei einer in der Öffentlichkeit stehenden Person der Fall ist. Sicherlich wissen viele Menschen, welche Zahnpasta Elvis benutzt hat, und jeder Programmierer kann selbst dann auf ein öffentliches Feld verweisen, wenn es nicht *Elvis* heißt.

In Java werden Wörter wie `public` *Zugriffsmodifizierer* genannt. Sicherlich haben Sie schon Felder und Methoden gesehen, die in ihren Deklarationen keinen Zugriffsmodifizierer enthielten. Von einem Feld oder einer Methode dieser Art wird gesagt, dass sie über einen *Standardzugriff* verfügen. Viele Beispiele in diesem Buch verwenden den Standardzugriff, ohne dies an die große Glocke zu hängen. In einigen Kapiteln geht das auch so in Ordnung, aber nicht in diesem hier. Ich beschreibe in diesem Kapitel die zentralen Bestandteile des Standardzugriffs.

Außerdem lernen Sie einen weiteren Zugriffsmodifizierer kennen, der bisher in keinem Beispiel verwendet worden ist. (Zumindest erinnere ich mich nicht daran.) Es handelt sich dabei um den Zugriffsmodifizierer `protected`. Ja, dieses Kapitel behandelt auch die weniger erfreulichen Dinge, die sich hinter einem geschützten (was *protected* auf Deutsch bedeutet) Zugriff verbergen.

Klassen, Zugriff und Multipart-Programmierung

Bei all diesen Begriffen kann man sich leicht in der Terminologie verheddern, weshalb wir uns noch einmal um ein paar

Grundlagen kümmern müssen. (Die meisten Begriffe, die Sie benötigen, stammen aus [Kapitel 10](#), aber es lohnt sich, zu Beginn dieses Kapitels noch einmal einen Blick darauf zu werfen.) Hier kommt zunächst ein täuschend gut gemachtes Codestückchen.

```
class MyClass {  
    // ein Feld (ein Mitglied):  
    int myField;  
    // eine Methode (ein weiteres Mitglied):  
    void myMethod() {  
  
    // eine methodenlokale Variable (KEIN Mitglied):  
        int myOtherField;  
    }  
}
```

Die Kommentare in diesem Code sagen Ihnen, worum es geht. Hier gibt es zwei Arten von Variablen: Felder und methodenlokale Variablen. Dieses Kapitel handelt nicht von methodenlokalen Variablen. Es kümmert sich um Methoden und Felder.

Es ist ziemlich lästig, immer mit dem Ausdruck »Methoden und Felder« arbeiten zu müssen. Deshalb ist es einfacher, beide Begriffe unter einer Bezeichnung zusammenzufassen. Diese Bezeichnung lautet *Mitglieder* einer Klasse.

Mitglied oder Klassen

An dieser Stelle müssen Sie auf eine wichtige Unterscheidung achten. Denken Sie an das Java-Schlüsselwort `public`. Wie Sie vielleicht von früheren Kapiteln her wissen, können Sie es vor ein Mitglied setzen. Sie können

```
public static void main(String args[]) {  
    oder  
  
public amountInAccount = 50.22;  
    schreiben
```

Diese Art, das Schlüsselwort `public` zu verwenden, ist wohl kaum eine Überraschung. Was Sie aber vielleicht noch nicht wissen, ist, dass Sie das Schlüsselwort `public` auch vor eine Klasse setzen können. So können Sie zum Beispiel schreiben:

```
public class Drawing {
    // Hier steht Ihr Code
}
```

Das Java-Schlüsselwort `public` hat zwei sich leicht unterscheidende Bedeutungen – eine für Mitglieder und eine andere für Klassen. Der größte Teil dieses Kapitels handelt von der Bedeutung, die `public` (und andere Schlüsselwörter) für Mitglieder haben. Nur der letzte Teil dieses Kapitels (mit der Überschrift *Zugriffsmodifizierer für Java-Klassen*) kümmert sich um die Bedeutung von `public` für Klassen.

Zugriffsmodifizierer für Mitglieder

Dieses Kapitel handelt von Mitgliedern. Dies bedeutet aber nicht, dass Sie Java-Klassen ignorieren können. Mitglied oder nicht, der Ort, an dem alle Aktionen stattfinden, ist immer noch die Java-Klasse. Jedes Feld wird in einer bestimmten Klasse deklariert, gehört zu dieser Klasse und ist ein Mitglied dieser Klasse.

Dasselbe gilt für Methoden. Jede Methode wird in einer bestimmten Klasse deklariert, gehört zu dieser Klasse und ist ein Mitglied dieser Klasse. Sind Sie in der Lage, den Namen eines bestimmten Mitglieds an einer beliebigen Stelle in Ihrem Code zu verwenden? Um mit der Beantwortung dieser Frage zu beginnen, prüfen Sie zunächst, ob sich diese Stelle innerhalb oder außerhalb der Klasse des Mitglieds befindet:

✓ Wenn das Mitglied privat ist, kann nur Code direkt auf den Namen des Mitglieds verweisen, der sich innerhalb der Klasse des Mitglieds befindet.

```
class SomeClass {
    private int myField = 10;
}

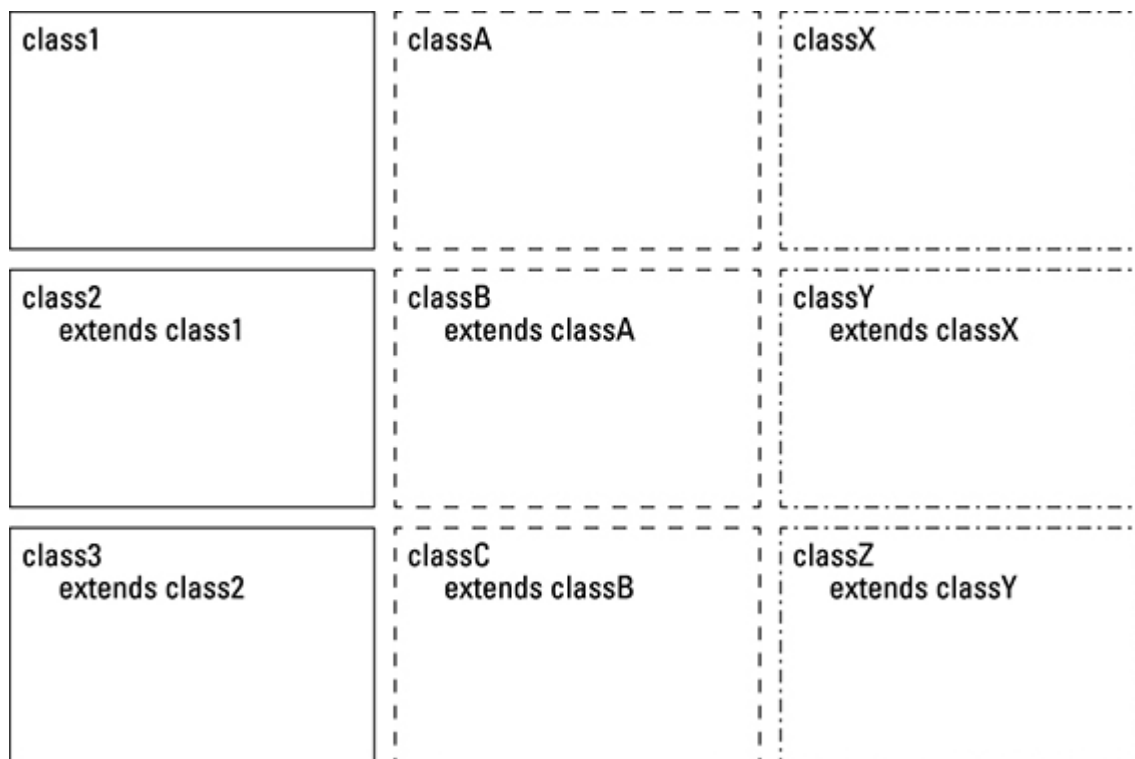
class SomeOtherClass {
    public static void main(String args[]) {
        SomeClass someObject = new SomeClass();

        //Dies hier funktioniert nicht:
        System.out.println(someObject.myField);
    }
}
```

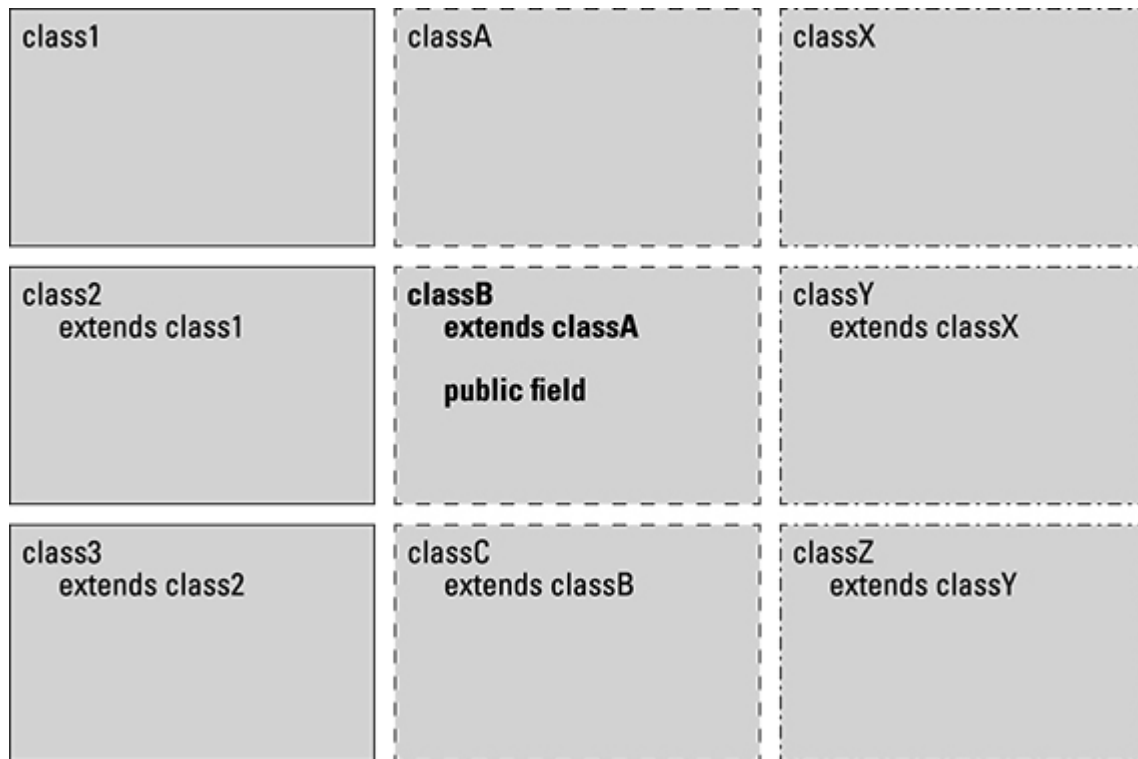
- ✓ Wenn das Mitglied öffentlich ist, kann jeder Code direkt auf den Namen verweisen.

```
class SomeClass {  
    public int myField = 10;  
}  
  
class SomeOtherClass {  
    public static void main(String args[]) {  
        SomeClass someObject = new SomeClass();  
  
        //Dies funktioniert:  
        System.out.println(someObject.myField);  
    }  
}
```

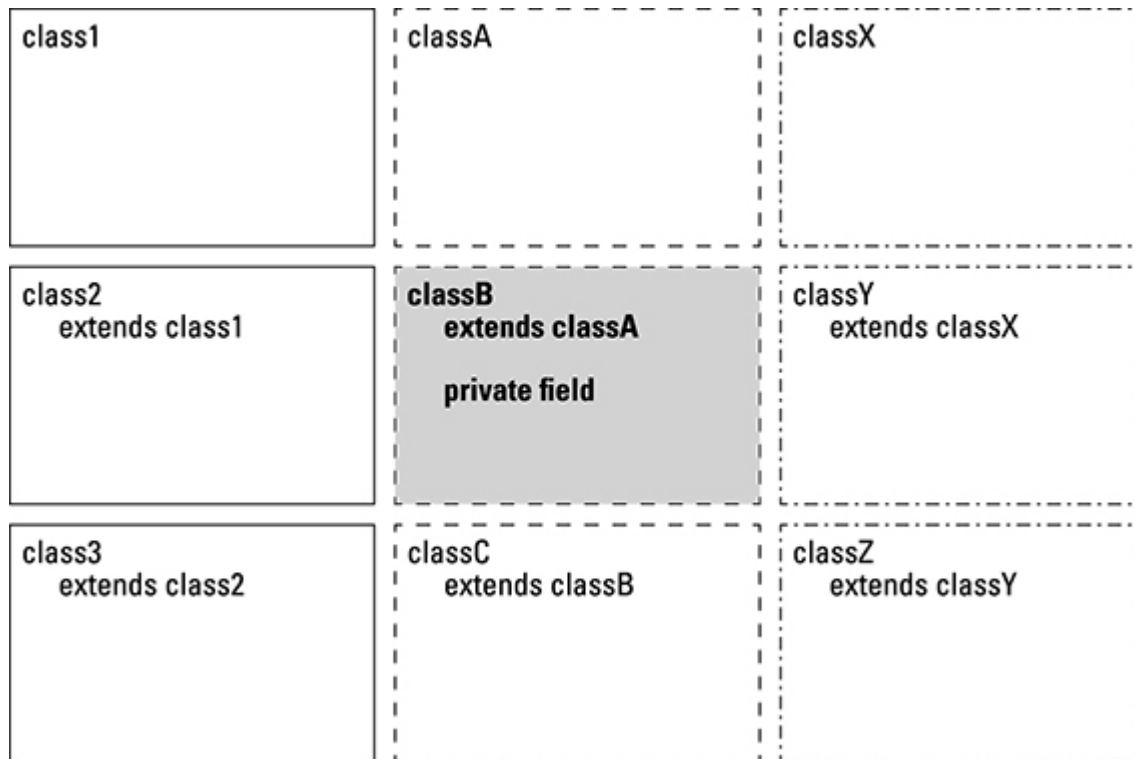
[Abbildung 14.1](#), [Abbildung 14.2](#) und [Abbildung 14.3](#) erläutern die Ideen auf eine etwas andere Weise.



[Abbildung 14.1](#): Mehrere Klassen und ihre Unterklassen



[Abbildung 14.2](#): Der Codebereich, in dem ein öffentliches Feld oder eine öffentliche Methode verwendet werden kann (schattiert)



[Abbildung 14.3](#): Der Codebereich, in dem ein privates Feld oder eine private Methode verwendet werden kann (schattiert)

Eine Zeichnung auf einem Frame platzieren

Damit die Sache mit dem Zugriffsmodifizierer klarer wird, brauchen Sie noch ein oder zwei Beispiele. Im ersten Beispiel dieses Kapitels ist so gut wie alles öffentlich. Bei einem öffentlichen Zugriff müssen Sie sich keine Gedanken darüber machen, wer was nutzen kann.

Der Code dieses ersten Beispiels besteht aus mehreren Teilen. Der erste Teil, der sich in [Listing 14.1](#) befindet, zeigt einen `ArtFrame` an. Auf der Oberfläche dieses Frames befindet sich ein `Drawing` (deutsch *Zeichnung*). Wenn sich alle Teile an Ort und Stelle befinden, zeigt der Code in [Listing 14.1](#) ein Fenster wie das in [Abbildung 14.4](#) an.

```
import com.burdbrain.drawings.Drawing;
import com.burdbrain.frames.ArtFrame;
class ShowFrame {
```

```

    public static void main(String args[]) {
        ArtFrame artFrame = new ArtFrame(new Drawing(
    ));
        artFrame.setSize(200, 100);
        artFrame.setVisible(true);
    }
}

```

[Listing 14.1](#): Einen Frame anzeigen



[Abbildung 14.4](#): Ein *ArtFrame*

Der Code in [Listing 14.1](#) erstellt eine neue Instanz von `ArtFrame`. Sie könnten vermuten, dass `ArtFrame` eine Unterklasse einer Java-Klasse `Frame` sei, was auch richtig ist. In [Kapitel 9](#) steht, dass Java-Frames standardmäßig unsichtbar sind. Deshalb rufen Sie in [Listing 14.1](#) die Methode `setVisible` auf, um die Instanz von `ArtFrame` sichtbar werden zu lassen.

Beachten Sie nun, dass [Listing 14.1](#) mit zwei `import`-Deklarationen beginnt. Die erste dieser beiden Deklarationen erlaubt es Ihnen, den Namen `Drawing` aus dem Paket `com.burdbrain.drawings` in seiner Kurzform zu verwenden. Aufgrund der zweiten `import`-Deklaration sind Sie in der Lage, den Namen `ArtFrame` aus dem Paket `com.burdbrain.frames` in seiner Kurzform zu verwenden. (Sie finden in [Kapitel 9](#) Informationen zu `import`-Deklarationen.)

Wenn Sie detektivisch veranlagt sind, fragen Sie sich vielleicht: »Er muss noch mehr Code geschrieben haben (Code, den ich hier nicht sehe) und diesen Code in Paketen untergebracht haben, die er `com.burdbrain.drawings` und `com.burdbrain.frames` genannt hat.« Und Sie haben recht. Damit [Listing 14.1](#) funktioniert, erstelle ich etwas, das *Drawing*

(deutsch *Zeichnung*) genannt wird. Meine Drawings bringe ich dann im Paket `com.burdbrain.drawings` unter. Außerdem benötige ich eine Klasse `ArtFrame`, die ich, wie alle diese Klassen, im Paket `com.burdbrain.frames` ablege.

Was ist nun ein `Drawing` wirklich? Wenn Sie so daran interessiert sind, das in Erfahrung zu bringen, schauen Sie sich [Listing 14.2](#) an.

```
package com.burdbrain.drawings;
import java.awt.Graphics;
public class Drawing {
    public int x = 40, y = 40, width = 40, height = 40;
    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

[Listing 14.2](#): Die Klasse `Drawing`

Der Code für die Klasse `Drawing` ist ziemlich schlank. Er enthält ein paar `int`-Felder und die Methode `paint`. Das ist alles. Wenn ich eine Klasse dieser Art erstelle, versuche ich, sie schlank zu halten.

Hier einige Anmerkungen zur Klasse `Drawing`:

✓ **Oben im Code steht eine *Paketdeklaration*.** Sieh da! Ich habe dafür gesorgt, dass meine Klasse `Drawing` zu einem Paket gehört – dem Paket `com.burdbrain.drawings`. Ich habe den Namen dieses Pakets nicht an den Haaren herbeigezogen. Die Konventionen (die von denen erlassen worden sind, die Java entwickelt haben) sagen, dass Sie den Namen eines Pakets mit dem umgekehrten Namen Ihrer Domäne beginnen. Deshalb habe ich `burdbrain.com` umgekehrt und zu `com.burdbrain` gemacht. Dann fügen Sie einen oder mehrere beschreibende Bezeichnungen hinzu, die durch Punkte voneinander getrennt werden. Ich habe *drawings* hinzugefügt, weil ich vorhabe, in diesem Paket meine Zeichnungen (englisch *Drawings*) abzulegen.

- ✓ **Die Klasse `Drawing` ist öffentlich.** Eine öffentliche Klasse kann von außen angegriffen werden. Deshalb vermeide ich es in der Regel, das Schlüsselwort `public` vor eine Klasse zu setzen. Aber in [Listing 14.1](#) bin ich gezwungen, meine Klasse `Drawing` öffentlich zu deklarieren. Wenn ich das nicht mache, sind Klassen, die sich nicht im Paket `com.burdbrain` befinden, nicht in der Lage, die schönen Dinge aus [Listing 14.2](#) zu nutzen. Gerade die Zeile

```
ArtFrame artFrame = new ArtFrame(new Drawing());
```

ist in [Listing 14.1](#) ohne die öffentliche Klasse `Drawing` nicht legal.

- ✓ **Der Code enthält eine Methode `paint`.** Diese Methode verwendet einen Standardtrick, um etwas auf dem Bildschirm erscheinen zu lassen. Der Parameter `g` in [Listing 14.2](#) wird *Grafikpuffer* genannt. Damit etwas erscheint, müssen Sie nur noch in diesem Grafikpuffer zeichnen, der dann auf dem Bildschirm des Computers ausgegeben wird.

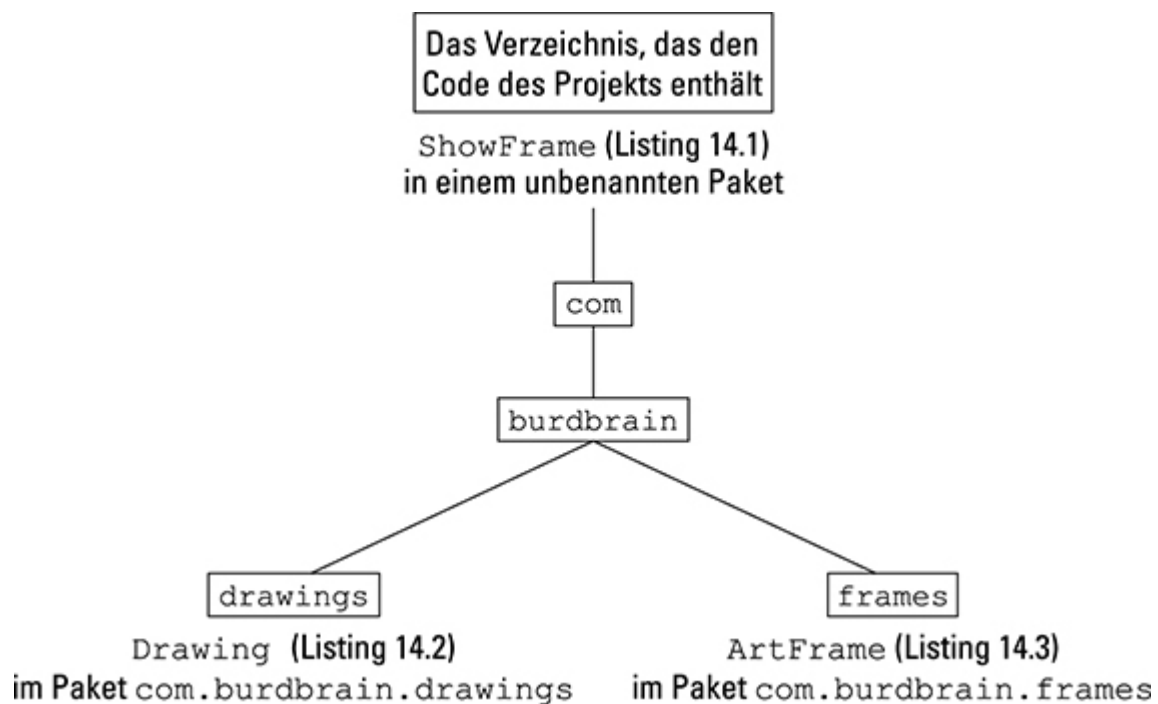
Hier ein paar Einzelheiten zu diesem Thema: In [Listing 14.2](#) enthält die Methode `paint` einen Parameter `g`. Dieser Parameter verweist auf eine Instanz der Klasse `java.`

`awt.Graphics`. Da eine Instanz von `Graphics` als Puffer dient, werden die Dinge, die Sie in diesem Puffer platzieren, letztendlich auf dem Bildschirm angezeigt. Dieser Puffer kennt, wie alle Instanzen der Klasse `java.awt.Graphics`, mehrere Methoden – von denen eine `drawOval` ist. Wenn Sie diese Methode aufrufen, geben Sie eine Anfangsposition an (`x` Pixel vom linken Rand des Frames und `y` Pixel vom oberen Rand des Frames). Außerdem legen Sie die Gesamtgröße der Zeichnung fest, indem Sie in den Parametern `width` (*Breite*) und `height` (*Höhe*) Pixelwerte hinterlegen. Der Aufruf der Methode `drawOval` bringt im `Graphics`-Puffer ein kleines rundes Ding unter. Dann werden der `Graphics`-Puffer, das runde Ding und was sonst noch vorhanden ist, auf dem Bildschirm angezeigt.

Die Verzeichnisstruktur

Der Code in [Listing 14.2](#) gehört zum Paket `com.burdbrain.drawings`. Wenn Sie eine Klasse in ein Paket legen, müssen Sie eine Verzeichnisstruktur erstellen, die den Namen des Pakets widerspiegelt.

Um Code unterzubringen, der zum Paket `com.burdbrain.drawings` gehören soll, benötigen Sie drei Verzeichnisse: ein Verzeichnis `com`, ein Unterverzeichnis von `com` mit dem Namen `burdbrain` und ein Unterverzeichnis von `burdbrain` mit dem Namen `drawings`. [Abbildung 14.5](#) zeigt die gesamte Verzeichnisstruktur.



[Abbildung 14.5](#): Die Dateien und Verzeichnisse Ihres Projekts



Wenn sich Ihr Code nicht in dem entsprechenden Verzeichnis befindet, erhalten Sie einen mehr als unerfreulichen `NoClassDefFoundError`. Glauben Sie mir, es macht niemals Spaß, diesen Fehler zu Gesicht zu bekommen. Wenn Sie diesen Fehler sehen, gibt es keine Hinweise

darauf, wo die fehlende Klasse sein könnte oder wo der Compiler sie erwartet. Wenn Sie ruhig bleiben, schaffen Sie es, die nötigen Informationen selbstständig zusammenzusuchen, aber wenn Sie in Panik geraten, stochern Sie stundenlang im Nebel herum. Als erfahrener Java-Programmierer erinnere ich mich an viele Blessuren, die dieser `NoClassDefFoundError` zu verantworten hat.

An den richtigen Stellen nach Dateien suchen

Sie versuchen, das Programm in [Listing 14.1](#) zu kompilieren. Der Java-Compiler kämpft sich durch den Code und stolpert über fehlende Teile. Da ist zunächst einmal das Ding mit dem Namen `ArtFrame`. Und die Sache mit `Drawing`. [Listing 14.1](#) definiert eine Klasse, die `ShowFrame` und nicht `ArtFrame` oder `Drawing` heißt. Wo soll sich nun der Compiler Informationen über `ArtFrame` und `Drawing` herholen?

Dieses Problem kann richtiggehend entmutigen. Sollte der Computer Ihre gesamte Festplatte nach Dateien wie `ArtFrame.java` oder `Drawing.class` absuchen müssen? Wie groß ist Ihre neue Festplatte? 400 GB? 750 GB? 6.000.000 GB? Und wie sieht das mit Verweisen auf Dateien aus, die sich auf Netzwerklaufwerken befinden? Das Suchgebiet könnte unendlich groß werden. Und was ist, wenn der Compiler schließlich alle offenen Punkte abgearbeitet hat? Dann versuchen Sie, Ihren Code ablaufen zu lassen, und die Java Virtual Machine (JVM) beginnt mit der Suche. (Informationen über die Java Virtual Machine stehen in [Kapitel 2](#).)

Um dieses Problem in den Griff zu bekommen, definiert Java etwas, das sich `CLASSPATH` nennt. Es gibt Programmierer, die solch einen `CLASSPATH` jedes Mal neu vorgeben, wenn sie ein Java-Programm ausführen. Andere legen eine systemweite `CLASSPATH`-Variable an. (Wenn Sie sich mit der Variablen `PATH` auf Windows- und Unix-Computern auskennen, wissen Sie vielleicht auch schon, wie so etwas geht.) Auf jeden Fall benötigen der Compiler und die JVM eine Liste mit den Orten, an denen sie nach Code Ausschau halten können. Ohne eine solche Liste schauen diese Java-Werkzeuge nämlich nirgendwo nach und finden Klassen wie `ArtFrame` oder `Drawing` nicht. Sie erhalten dann eine Nachricht mit einem Kann-ich-nicht-finden-Symbol oder eine `NoClassDefFoundError`-Meldung und Ihr Glück ist vorbei.

Einen Frame erstellen

Die ersten drei Listings dieses Kapitels entwickeln ein Beispiel, das aus mehreren Teilen besteht. Dieser Abschnitt hier kümmert sich um das letzte dieser drei Teile. Es ist für das Verständnis von Zugriffsmodifizierern, dem zentralen Thema dieses Kapitels, nicht von entscheidender Bedeutung. Wenn Sie also keine Lust haben, sich mit den Erklärungen zu [Listing 14.3](#) abzugeben, überspringen Sie diesen Abschnitt einfach. Ansonsten sollten Sie weiterlesen, wenn Sie mehr über die Java-Klasse `Swing` in Erfahrung bringen wollen.

```
package com.burdbrain.frames;
import java.awt.Graphics;
import javax.swing.JFrame;
import com.burdbrain.drawings.Drawing;
public class ArtFrame extends JFrame {
    private static final long serialVersionUID = 1L;
    Drawing drawing;
    public ArtFrame(Drawing drawing) {
        this.drawing = drawing;
        setTitle("Moderne Kunst");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
    public void paint(Graphics g) {
        drawing.paint(g);
    }
}
```

[Listing 14.3](#): Die Klasse `ArtFrame`

[Listing 14.3](#) verfügt über alle Einzelheiten, die Sie benötigen, um eine Zeichnung in einem Java-Frame zu platzieren. Der Code verwendet mehrere Namen aus der Java-API. Ich erkläre die meisten dieser Namen in den [Kapiteln 9](#) und [10](#).

In [Listing 14.3](#) gibt es nur einen einzigen neuen Namen: `paint`. Die Methode verweist auf eine andere `paint`-Methode, die zum Objekt `Drawing` gehört. Das Objekt `ArtFrame` erstellt auf dem Bildschirm Ihres Computers ein schwebendes Fenster. Was in diesem Fenster gezeichnet wird, hängt davon ab, welches `Drawing`-Objekt an den Konstruktor von `ArtFrame` übergeben wird.

Wenn Sie sich noch einmal mit den Listings 14.1 bis 14.3 beschäftigen, fällt Ihnen vielleicht etwas Eigenartiges auf. Die Methode `paint` aus [Listing 14.3](#) scheint niemals aufgerufen zu werden. Nun, in vielen Fällen, die mit dem Erzeugen von Fenstern (in Java *Frames* genannt) zu tun haben, deklarieren Sie eine Methode `paint`, die dann einfach nur ruhig im Code vor sich hin schlummert. Wenn dann das Programm ausgeführt wird, ruft der Computer diese Methode `paint` automatisch auf.

Das ist die Geschichte der `javax.swing.JFrame`-Objekte. In [Listing 14.3](#) wird die Methode `paint` des Frames im Hintergrund aufgerufen. Dann ruft die Methode `paint` des Frames die Methode `paint` des `Drawing`-Objekts auf, die dann ein Oval auf dem Frame zeichnet. Und das ist dann das, was Sie in [Abbildung 14.4](#) sehen.

Sich heimlich vom ursprünglichen Code absetzen

Der von Ihnen gerne gesehene Softwarevertrieb Burd Brain Consulting hat Ihnen zwei Dateien verkauft: `Drawing.class` und `ArtFrame.class`. Sie als Kunden sind nicht in der Lage, den Code in diesen beiden Dateien zu sehen. Ihnen bleibt also nichts anderes übrig, als mit dem klarzukommen, was in diesen beiden Dateien geschieht. Sie möchten nun aber unbedingt Einfluss darauf nehmen, wie das Oval in [Abbildung 14.4](#) aussieht – es soll ein wenig größer werden. Zu diesem Zweck erstellen Sie eine Unterklasse der Klasse `Drawing` – die Klasse `DrawingWide` – und bauen sie in [Listing 14.4](#) ein.

```
import java.awt.Graphics;
import com.burdbrain.drawings.Drawing;
public class DrawingWide extends Drawing {
    int width = 100, height = 30;
    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

[Listing 14.4](#): Eine Unterklasse der Klasse `Drawing`

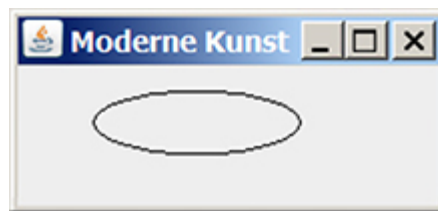
Um den Code aus [Listing 14.4](#) verwenden zu können, müssen Sie eine Zeile in [Listing 14.1](#) ändern. Machen Sie sie zu

```
ArtFrame artFrame = new ArtFrame(new DrawingWide() );
```

Außerdem können Sie in [Listing 14.1](#) die `import`-Deklaration `com.burdbbrain. drawings.Drawing` entfernen, weil sie nicht mehr benötigt wird.

[Listing 14.4](#) definiert eine Unterklasse der Klasse `Drawing`. Sie überschreiben in dieser Unterklasse die ursprünglichen Felder `width` und `height` und die ursprüngliche Methode `paint`.

[Abbildung 14.6](#) zeigt den Frame, den Sie nun erhalten.



[Abbildung 14.6](#): Ein anderer `ArtFrame`

Vielleicht fällt Ihnen im Vorübergehen auf, dass der Code in [Listing 14.4](#) nicht mit einer Paketdeklaration beginnt. Dies bedeutet, dass Ihre gesamte Dateisammlung aus den folgenden drei Paketen stammt:

- ✓ **Dem Paket `com.burdbbrain.drawings`.** In diesem Paket befindet sich die ursprüngliche Klasse `Drawing` aus [Listing 14.2](#).
- ✓ **Dem Paket `com.burdbbrain.frames`.** Aus diesem Paket stammt die Klasse `ArtFrame`.
- ✓ **Einem immer vorhandenen, unbenannten Paket.** Wenn Sie in Java eine Datei ohne eine Paketdeklaration beginnen, landet der gesamte Code in einem großen unbenannten Paket. Die [Listings 14.1](#) und [14.4](#) befinden sich in demselben unbenannten Paket. Letztendlich ist es so, dass die meisten

Listings aus den ersten 12 Kapiteln dieses Buches in Javas unbenanntem Paket liegen.

Zum jetzigen Zeitpunkt enthält Ihr Projekt zwei »zeichnende« Klassen – die ursprüngliche Klasse `Drawing` und die neue Klasse `DrawingWide`. So ähnlich diese beiden Klassen auch sein mögen, so befinden sie sich doch in zwei verschiedenen Paketen. Das sollte eigentlich niemanden überraschen. Die Klasse `Drawing`, die von Ihren Freunden bei Burd Brain Consulting entwickelt worden ist, befindet sich in einem Paket, dessen Namen mit *com.burdbrain* beginnt. Da Sie `DrawingWide` selbst erstellt haben, hat diese Klasse nichts im Paket `com.burdbrain` zu suchen.

Es wäre sehr gescheit, `DrawingWide` in einem Ihrer Pakete, zum Beispiel in `de.meinedomäne.zeichnungen`, unterzubringen, aber für den Anfang reicht das unbenannte Paket aus.

Auf jeden Fall wird Ihre Klasse `DrawingWide` kompiliert und kann – wie geplant – ausgeführt werden.

Standardzugriff

Wenn Sie den Text bis hierhin in seiner vorgegebenen Reihenfolge gelesen haben, wissen Sie, dass das letzte Beispiel sehr glücklich endet. Der Code in [Listing 14.4](#) funktioniert wunderbar. Aber halt! Was wäre, wenn ... Ich drehe in diesem Kapitel die Zeit ein wenig zurück und zeige Ihnen, was geschehen wäre, wenn Sie im Code von [Listing 14.2](#) ein Wort vergessen hätten.

Der Umgang mit verschiedenen Versionen eines Programms kann schwindelerregend sein. Aus diesem Grund beginne ich diese Diskussion mit einer Beschreibung dessen, was Sie haben. Als Erstes besitzen Sie eine Klasse `Drawing`. Die Felder dieser Klasse werden als öffentliche Felder mit Standardzugriff deklariert. Die Klasse `Drawing` befindet sich im Paket `com.burdbrain.drawings` (siehe Listing 4.5).

```
package com.burdbrain.drawings;  
import java.awt.Graphics;
```

```
public class Drawing {
    int x = 40, y = 40, width = 40, height = 40;
    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

[Listing 14.5](#): Felder mit Standardzugriff

Als Zweites haben Sie die Unterklasse `DrawingWide` (die, damit es für Sie bequemer ist, in [Listing 14.6](#) kopiert wurde).

```
import com.burdbrain.drawings.*;
import java.awt.Graphics;
public class DrawingWide extends Drawing {
    int width = 100, height = 30;
    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

[Listing 14.6](#): Der fehlgeschlagene Versuch, eine Unterklasse zu erstellen

Das Problem ist, dass das ganze Ding scheitert. Der Code in [Listing 14.6](#) wird nicht kompiliert. Sie erhalten stattdessen diese Fehlermeldung:

```
x is not public in com.burdbrain.drawings.Drawing;
cannot be accessed from outside package
y is not public in com.burdbrain.drawings.Drawing;
cannot be accessed from outside package
```

Sie besagt, dass sowohl `x` als auch `y` nicht öffentlich deklariert wurde und damit ein Zugriff von außerhalb des Pakets nicht möglich ist.

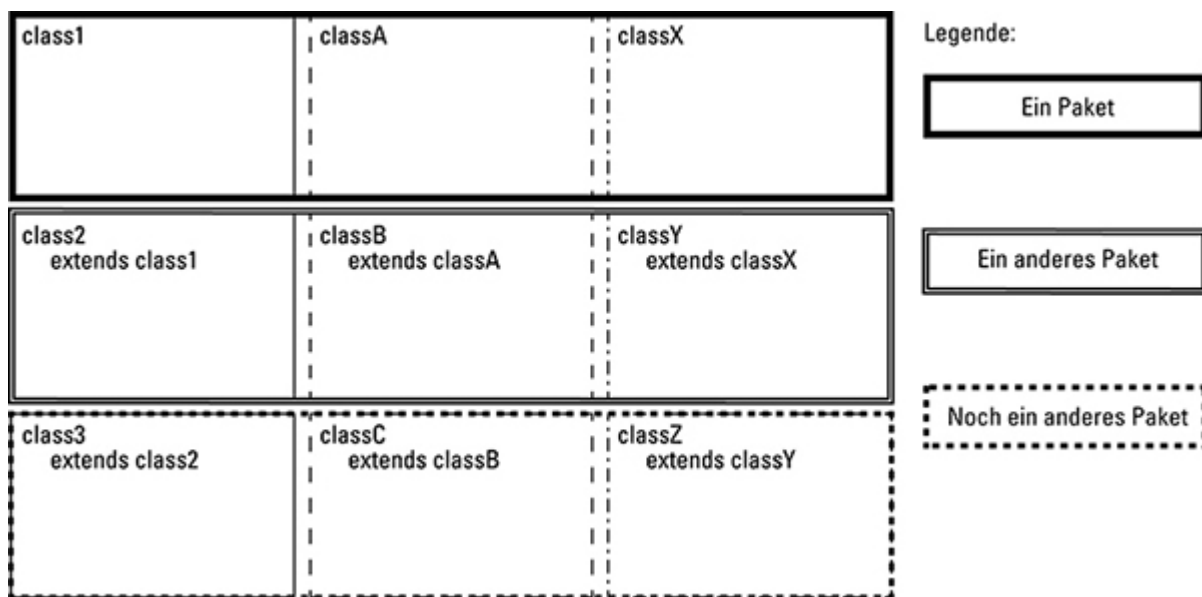
Der Code lässt sich nicht kompilieren, weil auf Felder mit einem Standardzugriff nicht direkt von außerhalb ihres Pakets verwiesen werden kann. Dies gilt sogar für eine Unterklasse der Klasse, in der sich diese Felder befinden. Dasselbe gilt für Methoden, die nur über einen Standardzugriff verfügen.



Die Zugriffsregeln, die ich in diesem Kapitel beschreibe, gelten nicht für methodenlokale Variablen. Auf eine

methodenlokale Variable kann nur innerhalb ihrer Methode zugegriffen werden. (Ich beschreibe diesen Variablentyp in [Kapitel 10.](#))

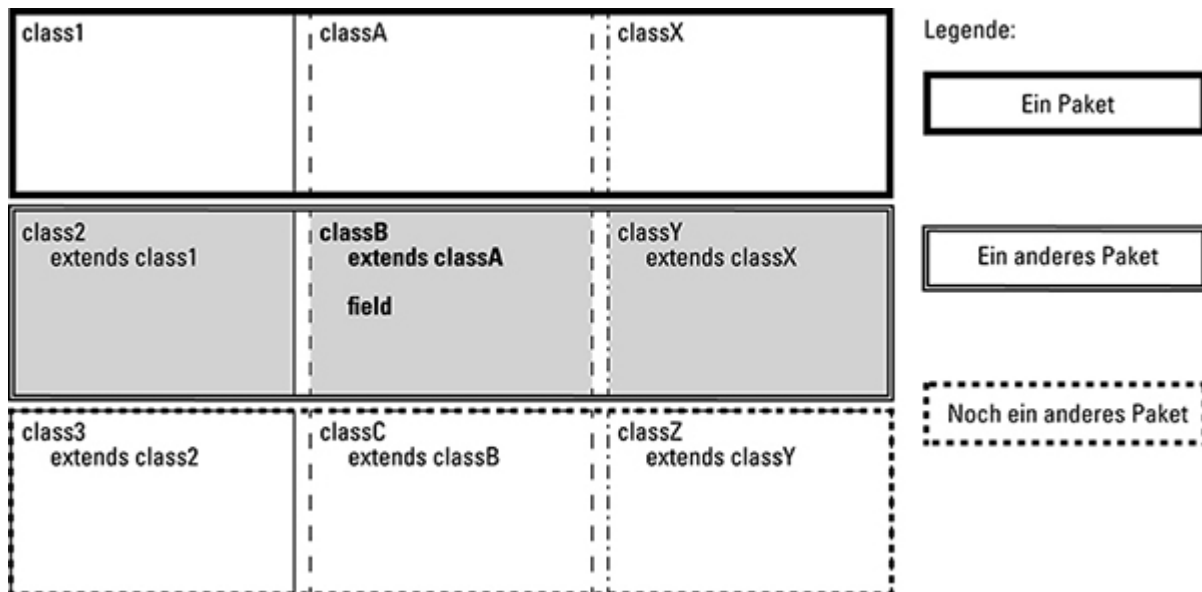
In Java ist der Standardzugriff auf ein Mitglied einer Klasse ein paketweiter Zugriff. Sie können auf ein Mitglied, das ohne das Wort `public`, `private` oder `protected` deklariert wird, nur innerhalb des Paketes zugreifen, zu dem die Klasse gehört. [Abbildung 14.7](#) und [Abbildung 14.8](#) verdeutlichen dies.



[Abbildung 14.7](#): Auf Unterklassen zugeschnittene Pakete

Wieder ins Paket zurückkrabbeln

Eigentlich liebe ich es, Post zu bekommen. Im schlimmsten Fall handelt es sich um Junk-Mail, die ich sofort in den Papierkorb schmeiße. Am liebsten habe ich Dinge, mit denen ich etwas anfangen kann, wie ein neues Spielzeug oder etwas, das jemand nur für mich gebastelt hat.



[Abbildung 14.8](#): Der Codebereich, in dem ein Standardfeld oder eine Standardmethode verwendet werden kann

Heute ist mein Glückstag. Irgendjemand von Burd Brain Consulting hat mir eine Unterklasse der Klasse `Drawing` geschickt. Dabei handelt es sich im Prinzip um denselben Code wie in Listing 14.6. Der einzige Unterschied ist, dass sich diese neue Klasse `DrawingWideBB` im Paket `com.burdbrain.drawings` befindet. [Listing 14.7](#) stellt diesen Code dar.

```
package com.burdbrain.drawings;
import java.awt.Graphics;
public class DrawingWideBB extends Drawing {
    int width = 100, height = 30;
    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

[Listing 14.7](#): Jawohl, das ist eine Unterklasse.

Damit ich diesen Code auch ausführen kann, muss ich in [Listing 14.1](#) eine Zeile ändern:

```
ArtFrame artFrame = new ArtFrame(new DrawingWideBB())
;
```

Wenn Sie [Listing 14.7](#) in Verbindung mit der Klasse `Drawing` in [Listing 14.5](#) ausführen, funktioniert alles richtig gut. Warum das so ist? `Drawing` und `DrawingWideBB` befinden sich in demselben Paket. Schauen Sie sich noch einmal [Abbildung 14.8](#) an und achten Sie auf den schattierten Bereich, der ein ganzes Paket umfasst. Der Code in der Klasse `DrawingWideBB` hat alle Rechte, um die Felder `x` und `y` zu nutzen, die mit Standardzugriff in der Klasse `Drawing` definiert worden sind.



Wenn Sie in [Listing 14.7](#) die Klasse `DrawingWideBB` verwenden wollen, müssen Sie in [Listing 14.1](#) eine Änderung vornehmen. Passen Sie die erste `import`-Deklaration so an, dass sie `import com.burdbrain.drawings.DrawingWideBB` lautet. Ändern Sie auch den Aufruf des Konstruktors des Objekts `ArtFrame` in `new ArtFrame(new DrawingWideBB())`.

Geschützter Zugriff

Als ich Java kennenlernte, glaubte ich, dass der Begriff *geschützt* so etwas wie *sicher* oder »Dieses Feld ist geschützt. Es muss schwer sein, darauf zuzugreifen.« bedeutet. Diese Vermutung stellte sich als falsch heraus. In Java ist ein Mitglied, das geschützt (*protected*) ist, weniger verborgen, weniger sicher und leichter zu verwenden, als wenn es über einen Standardzugriff erreichbar wäre. Dieses Konzept ist ziemlich seltsam.

Stellen Sie sich einen geschützten Zugriff so vor: Sie beginnen mit einem Feld, für das der Standardzugriff gilt (ein Feld, dem in der Deklaration das Wort `public`, `private` oder `protected` fehlt). Auf dieses Feld kann nur innerhalb des Pakets zugegriffen werden, zu dem es gehört. Fügen Sie nun an den Anfang der Deklaration dieses Feldes das Wort `protected` hinzu. Plötzlich können auch Klassen außerhalb des Pakets dieses Feldes darauf zugreifen. Sie sind nun in der Lage, aus einer Unterklasse heraus (der Klasse, zu der das Feld gehört) auf das Feld zu verweisen.

Ein solcher Verweis ist auch aus Unter-Unterklassen, Unter-Unter-Unterklassen und so weiter heraus möglich. Hier können Sie jeden Abkömmling der Deklarationsklasse des Feldes verwenden. Die [Listings 14.8](#) und [14.9](#) enthalten ein entsprechendes Beispiel.

```
package com.burdbrain.drawings;
import java.awt.Graphics;
public class Drawing {
    protected int x = 40, y = 40, width = 40, height
= 40;
    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

[Listing 14.8](#): Geschützte Felder

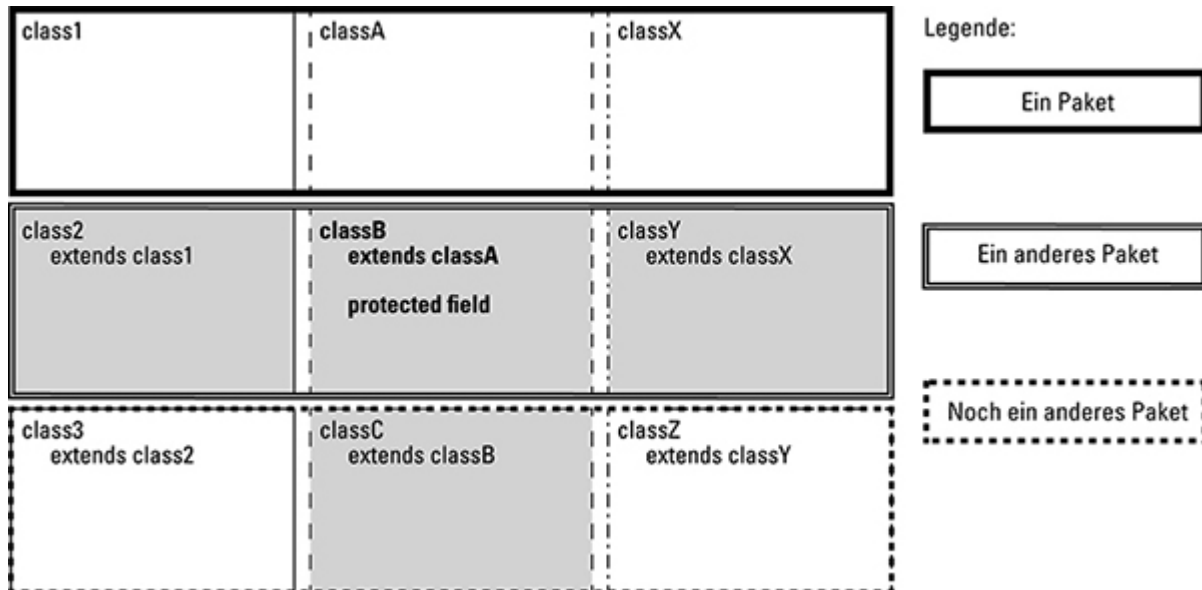
```
import java.awt.Graphics;
import com.burdbrain.drawings.Drawing;
public class DrawingWide extends Drawing {
    int width = 100, height = 30;
    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

[Listing 14.9](#): Die Unterklasse der Blauen Lagune, Teil II

[Listing 14.8](#) definiert die Klasse `Drawing`. [Listing 14.9](#) definiert `DrawingWide`, eine Unterklasse der Klasse `Drawing`.

In der Klasse `Drawing` sind die Felder `x`, `y`, `width` und `height` geschützt. Die Klasse `DrawingWide` verfügt über ihre eigenen Felder `width` und `height`, verweist aber auf die Felder `x` und `y`, die in der Elternklasse `Drawing` definiert werden. Dies bereitet selbst dann keine Probleme, wenn sich `DrawingWide` nicht in demselben Paket wie die Klasse `Drawing` befindet. (Die Klasse `Drawing` ist im Paket `com.burdbrain.drawings` enthalten; die Klasse `DrawingWide` gehört zum großen unbenannten Java-Paket.) Der Zugriff funktioniert, weil die Felder `x` und `y` in der Klasse `Drawing` geschützt sind.

Vergleichen Sie [Abbildung 14.8](#) mit [Abbildung 14.9](#). Beachten Sie die zusätzliche Schattierung in [Abbildung 14.9](#). Eine Unterklasse kann selbst dann auf ein geschütztes Mitglied einer Klasse zugreifen, wenn die Unterklasse zu einem anderen Paket gehört.



[Abbildung 14.9](#): Der Codebereich, in dem ein geschütztes Feld oder eine geschützte Methode verwendet werden kann (schattiert)



Arbeiten Sie in einem Team von Programmierern? Verwenden Personen, die nicht zu Ihrem Team gehören, eigene Paketnamen? In diesem Fall legen sie Unterklassen der von Ihnen definierten Klassen an, wenn sie Ihren Code verwenden wollen. Das ist dann der Moment, an dem der geschützte Zugriff ins Spiel kommt. Verwenden Sie diesen, wenn Sie wollen, dass auch andere Personen die Felder Ihres Codes direkt (von außerhalb) referenzieren können.

Klassen, die keine Unterklassen sind, in demselben Paket unterbringen

Diese Leute von Burd Brain Consulting senden Ihnen ein Stück Software nach dem anderen zu. Dieses Mal erhalten Sie eine Alternative zur Klasse `ShowFrame` – der Klasse in Listing 14.1. Diese neue Klasse `ShowFrameWideBB` gibt ein breiteres Oval wieder, ohne eine Unterklasse der alten Klasse `Drawing` zu erstellen. Stattdessen legt der neue `ShowFrameWideBB`-Code eine Instanz von `Drawing` an und ändert den Wert der Felder `width` und `height` der Instanz. [Listing 14.10](#) stellt den entsprechenden Code dar.

```
package com.burdbrain.drawings;
import com.burdbrain.frames.ArtFrame;
class ShowFrameWideBB {
    public static void main(String args[]) {
        Drawing drawing = new Drawing();
        drawing.width = 100;
        drawing.height = 30;
        ArtFrame artFrame = new ArtFrame(drawing);
        artFrame.setSize(200, 100);
        artFrame.setVisible(true);
    }
}
```

[Listing 14.10](#): Ein breiteres Oval zeichnen

Die Klasse `ShowFrameWideBB` aus [Listing 14.10](#) befindet sich in demselben Paket wie die Klasse `Drawing` (im Paket `com.burdbrain.drawings`). Aber `ShowFrameWideBB` ist keine Unterklasse der Klasse `Drawing`.

Stellen Sie sich nun vor, dass Sie die Klasse `ShowFrameWideBB` zusammen mit der Klasse `Drawing` aus [Listing 14.8](#) kompilieren – der Klasse, die die geschützten Felder enthält. Was geschieht dann? Alles läuft glatt ab, weil ein geschütztes Mitglied an zwei (nicht zusammenhängenden) Stellen zur Verfügung steht. Schauen Sie sich noch einmal [Abbildung 14.9](#) an. Ein geschütztes Mitglied steht nicht nur Unterklassen außerhalb des Pakets zur Verfügung, sondern grundsätzlich auch dem Code innerhalb des Pakets des Mitglieds (und zwar unabhängig davon, ob es sich bei diesem Code um eine Unterklasse handelt oder nicht).



[Listing 14.10](#) enthält eine Methode `main`, die sich innerhalb einer Klasse befindet, die wiederum Bestandteil eines Pakets (`com.burdbrain.drawings`) ist. Bei den meisten Entwicklungsumgebungen müssen Sie sich keine Gedanken machen, wenn Sie eine Methode `main` ausführen wollen, die aus einem benannten Paket stammt. Wenn Sie aber Programme aus der Befehlszeile heraus ablaufen lassen wollen, müssen Sie in der Regel einen vollqualifizierten Klassennamen eingeben. Um zum Beispiel den Code in [Listing 14.10](#) auszuführen, müssen Sie

```
java com.burdbrain.drawings.ShowFrameWideBB
```

schreiben.



Hinter dem geschützten Zugriff verbirgt sich weitaus mehr, als ich in diesem Abschnitt beschreibe. Die Java Language Specification geht spitzfindig auf Code ein, der mit der Implementierung eines Objekts zu tun hat. Wenn Sie als Neuling herausfinden wollen, wie in Java programmiert wird, machen Sie sich über diesen Punkt keine Gedanken. Kümmern Sie sich um so etwas erst, wenn Sie viele Java-Programme geschrieben haben. Wenn Sie dann über Fehlermeldungen stolpern, die besagen, dass eine Variable über einen geschützten Zugriff verfügt (`variable has protected access`), ist immer noch Zeit genug, sich darüber Gedanken zu machen. Wobei es sich dann aber doch lohnt, in der Java Language Specification einen Blick auf den Abschnitt über den geschützten Zugriff zu werfen. (Sie finden in [Kapitel 3](#) Informationen über die Java Language Specification.)

Zugriffsmodifizierer für Java-Klassen

Vielleicht verunsichern Sie die Dinge ein wenig, die Sie über Zugriffsmodifizierer für Mitglieder lesen. Schließlich ist in Java der Zugriff ein kompliziertes Thema mit vielen Drehungen und Wendungen. Aber die Unsicherheit sollte nun vorbei sein. Wenn Sie das Epos über Felder und Methoden mit der Geschichte über den Zugriff auf Klassen vergleichen, ist die Geschichte doch viel einfacher gestrickt als das Epos.

Eine Klasse kann entweder öffentlich oder nicht öffentlich sein. Wenn Sie etwas wie

```
public class Drawing
```

sehen, erblicken Sie die Deklaration einer öffentlichen Klasse. Demgegenüber bezeichnet

```
class ShowFrame
```

eine Klasse, deren Deklaration nicht öffentlich ist.

Öffentliche Klassen

Wenn eine Klasse öffentlich ist, können Sie von überall in Ihrem Code auf sie verweisen. Natürlich geht es aber auch dabei nicht ganz ohne Einschränkungen ab. Sie müssen allen Regeln gehorchen, die im Abschnitt *Die Verzeichnisstruktur* dieses Kapitels stehen. Außerdem müssen Sie sauber auf eine Klasse verweisen, die sich in einem Paket befindet. Sie können zum Beispiel in [Listing 14.1](#)

```
import com.burdbbrain.drawings.Drawing;
import com.burdbbrain.frames.ArtFrame;
...
```

```
ArtFrame artFrame = new ArtFrame(new Drawing());
```

schreiben oder Sie lassen die import-Deklarationen weg und schreiben

```
com.burdbbrain.frames.ArtFrame artFrame =
    new com.burdbbrain.frames.ArtFrame
        (new com.burdbbrain.drawings.Drawing());
```

Aber unabhängig davon, wie Sie vorgehen, muss der Code in beiden Fällen bestätigen, dass sich die Klassen `ArtFrame` und `Drawing` in benannten Paketen befinden.

Nicht öffentliche Klassen

Wenn eine Klasse nicht öffentlich ist, können Sie auf sie nur aus Code heraus verweisen, der sich im Paket der Klasse befindet.

Ich habe das ausprobiert. Zuerst bin ich zu [Listing 14.2](#) zurückgegangen und habe das Wort `public` gelöscht. Ich habe aus `public class Drawing` wieder das gute alte `class Drawing` gemacht:

```
package com.burdbrain.drawings;
import java.awt.Graphics;
class Drawing {
    public int x = 40, y = 40, width = 40, height = 40;
    public void paint(Graphics g) {
        g.drawOval(x, y, width, height);
    }
}
```

Dann habe ich den Code in [Listing 14.7](#) kompiliert.

Alles lief prima ab, weil [Listing 14.7](#) die folgenden Zeilen enthält:

```
package com.burdbrain.drawings;
public class DrawingWideBB extends Drawing
```

Da sich beide Codestücke in demselben Paket `com.burdbrain.drawings` befinden, war der Rückgriff auf die nicht öffentliche Klasse `Drawing` von der Klasse `DrawingWideBB` aus überhaupt kein Problem.

Aber dann versuchte ich, den Code in [Listing 14.3](#) zu kompilieren. Dieser Code beginnt mit

```
package com.burdbrain.frames;
```

Dieser Code befindet sich nicht im Paket `com.burdbrain.drawings`. Wenn der Computer in [Listing 14.3](#) die Zeile

```
Drawing drawing;
```

erreicht, fährt er im wahrsten Sinne des Wortes vor die Wand und gibt diese Meldung aus:

```
com.burdbrain.drawings.Drawing is not public
in com.burdbrain.drawings;
```

cannot be accessed from outside package

Diese Meldung besagt, dass von außen nicht auf das Paket zugegriffen werden kann, weil es nicht öffentlich ist.



Die Dinge sind niemals so einfach, wie es auf den ersten Blick aussieht. Die Regeln, die ich in diesem Abschnitt beschreibe, gelten für so gut wie jede Klasse in diesem Buch. Aber Java kennt noch ganz tolle Sachen, wie zum Beispiel *innere Klassen*, die ganz anderen Regeln folgen. Glücklicherweise haben die Programme eines Programmierneulings in der Regel wenig mit inneren Klassen zu tun. In diesem Buch kommen sie eigentlich nur in [Kapitel 14](#) vor (und ein paar verkleiden sich als `enum`-Typen). Aus diesem Grund können Sie für den Augenblick sehr glücklich mit den Regeln leben, die ich in diesem Abschnitt beschreibe.

15

Auf Tastatureingaben und Mausclicks reagieren

In diesem Kapitel ...

- ▶ Code erstellen, der mit Mausclicks (und ähnlichen Ereignissen) umgehen kann
 - ▶ Eine Oberfläche erstellen und verwenden
-

Ich kaufte meine erste Maus in den späten 1980er Jahren. Ich bezahlte 100 Dollar, und da ich die Maus nicht wirklich benötigte, habe ich vorher meine Frau um Erlaubnis gefragt. (Damals lief auf meinem Computer eine kombinierte Text/Windows-Umgebung. Alles, was ich mit einer Maus machen konnte, ließ sich auch ganz einfach über die Taste [Alt] erledigen.)

Jetzt haben wir das 21. Jahrhundert. Die letzten zehn Mäuse habe ich umsonst bekommen. Normale Mäuse fallen mir irgendwie in den Schoß. Ein paar exotische Mäuse habe ich im Computerladen gekauft. Eine kostete 10 Dollar, und es gab sie zusammen mit einem 10-Dollar-Gutschein.

Beim Schreiben dieses Kapitels verwende ich das neueste Stück meiner Maussammlung – die offizielle *Für-Dummies*-Maus. Diese gelb-weiße Schönheit enthält ein kleines mit Wasser gefülltes Fach. In meinem Fall umfließt das Wasser kein versunkenes Atlantis, sondern einen kleinen *Dummy-Mann* als Talisman. Das ist richtig niedlich. Es handelt sich um ein Geschenk, das mir Wiley Publishing gemacht hat.

Machen Sie weiter ... Klicken Sie auf diese Schaltfläche

Ich habe in früheren Kapiteln Fenster erstellt, die eigentlich nicht viel tun. Ein ganz normales Fenster zeigt zwar Informationen an, verfügt aber nicht über interaktive Elemente. Jetzt ist es an der Zeit, dies zu ändern. Das erste Beispiel dieses Kapitels ist ein Fenster, das eine Schaltfläche enthält. Wenn der Benutzer diese Schaltfläche anklickt, geschieht etwas. [Listing 15.1](#) zeigt den entsprechenden Code, und die Methode `main`, die diesen Code aufruft, befindet sich in [Listing 15.2](#).

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
class GameFrame extends JFrame implements ActionListener
{
    private static final long serialVersionUID = 1L;
    int randomNumber = new Random().nextInt(10) + 1;
    int numGuesses = 0;
    JTextField textField = new JTextField(5);
    JButton button = new JButton("Raten Sie");
    JLabel label = new JLabel(numGuesses + " Versuche"
);
    public GameFrame() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        add(textField);
        add(button);
        add(label);
        button.addActionListener(this);
        pack();
        setVisible(true);
    }
    @Override
    public void actionPerformed(ActionEvent e) {
```



```

        String textFieldText = textField.getText();
        if (Integer.parseInt(textFieldText)==randomNumb
er) {
            button.setEnabled(false);
            textField.setText(textField.getText() + " Ja
!");
            textField.setEnabled(false);
        } else {
            textField.setText("");
            textField.requestFocus();
        }
        numGuesses++;
        String guessWord =
            (numGuesses == 1) ? " Versuch" : " Versuche
";
        label.setText(numGuesses + guessWord);
    }
}

```

[Listing 15.1](#): Ein Ratespiel

```

class ShowGameFrame {

    public static void main(String args[]) {
        new GameFrame();
    }
}

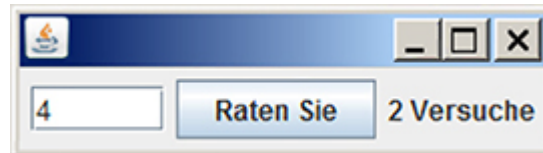
```

[Listing 15.2](#): Das Ratespiel starten

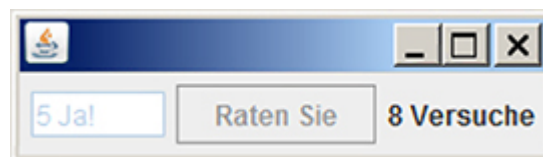
[Abbildung 15.1](#) und [Abbildung 15.2](#) spiegeln Schnappschüsse wider, die die Ausführung des Beispielsprogramms dieses Kapitels zeigen. Der Benutzer spielt in einem Fenster ein Ratespiel. Das Programm wählt im Hintergrund eine geheime Zahl (von 1 bis 10) aus. Dann zeigt das Programm ein Textfeld und eine Schaltfläche an. Der Benutzer gibt in das Textfeld eine Zahl ein und klickt auf die Schaltfläche. Nun passiert eines von zwei Dingen:

- ✓ **Wenn die Zahl, die der Benutzer eingegeben hat, nicht mit der geheimen Zahl übereinstimmt**, zeigt der Computer die Anzahl der bisherigen Rateversuche an, und der Benutzer kann erneut raten.

- ✓ **Wenn die Zahl, die der Benutzer eingegeben hat, mit der geheimen Zahl übereinstimmt**, werden im Textfeld die richtige Zahl und **Ja!** angezeigt. Damit ist das Spiel vorbei, und sowohl das Textfeld als auch die Schaltfläche werden deaktiviert. Beide Komponenten werden ausgegraut und reagieren weder auf Tastatureingaben noch auf Mausaktionen.



[Abbildung 15.1](#): Falsch geraten



[Abbildung 15.2](#): Endlich richtig geraten!

Der Code in [Listing 15.1](#), der den Frame, die Schaltfläche und das Textfeld erstellt, ist nichts Weltbewegendes. Ich habe in den [Kapiteln 9](#) und [10](#) ähnliche Dinge getan. Die Klasse `JTextField` ist neu, aber ein Textfeld unterscheidet sich nicht großartig von einer Schaltfläche oder einem Label. Die Klasse `JTextField` wird wie so viele andere Komponenten im Paket `javax.swing` definiert. Wenn Sie eine neue `JTextField`-Instanz anlegen, können Sie festlegen, wie viele Spalten das Feld erhalten soll. Ich erstelle in [Listing 15.1](#) ein Textfeld, das fünf Spalten breit ist. Außerdem verwendet dieses Listing einen nicht alltäglichen Operator, um sowohl für die Einzahl (`Versuch`) als auch die Mehrzahl (`Versuche`) gewappnet zu sein. Wenn Sie nicht sicher wissen, wie Sie mit dem Fragezeichen und dem Doppelpunkt umgehen müssen, werfen Sie einen Blick in [Kapitel 11](#).

Ereignisse und Ereignisbehandlung

Das wirklich Neue in [Listing 15.1](#) ist, wie damit umgegangen wird, wenn der Benutzer die Schaltfläche anklickt. Wenn Sie mit einer grafischen Oberfläche für Benutzer (*GUI* – für *Graphical User Interface*) arbeiten, wird alles, was ein Benutzer macht (zum Beispiel das Drücken einer Taste, das Bewegen der Maus, das Klicken mit der Maus oder was sonst noch ansteht), als *Ereignis* bezeichnet. Den Code, der auf das Drücken, Bewegen oder Klicken des Benutzers reagiert, nennt man *Ereignisbehandlung*.



Interface hat im Deutschen mehrere Bedeutungen. Es kann sowohl *Oberfläche* als auch *Schnittstelle* heißen. Und dann gibt es noch das Java-Objekt `Interface`, das niemals übersetzt wird. Achten Sie also darauf, in welchem Zusammenhang Sie *Interface* lesen.

[Listing 15.1](#) hat an drei Stellen mit dem Ereignis »Anklicken einer Schaltfläche« zu tun:

- ✓ Im oberen Teil der Deklaration der Klasse `GameFrame` heißt es, diese Klasse `implements ActionListener`.
- ✓ Der Konstruktor der Klasse `GameFrame` fügt der Liste mit den *Action Listeners* der Schaltfläche `this` hinzu.
- ✓ Der Code der Klasse `GameFrame` enthält eine Methode `actionPerformed`.

Diese drei Maßnahmen sorgen letztendlich dafür, dass die Klasse `GameFrame` in der Lage ist, auf Klickereignisse der Schaltfläche zu reagieren. Um zu verstehen, wie das funktioniert, müssen Sie wissen, was es mit einem *Interface* auf sich hat.

Das Java-Interface

Vielleicht ist Ihnen aufgefallen, dass Klassen in Java niemals mehr als eine Elternklasse erweitern können. Sie dürfen also niemals schreiben:

```
class NieMachen extends ErsteKlasse, ZweiteKlasse
```

Eine Klasse kann nur eine Elternklasse haben, und das geht auch in Ordnung, wenn Sie wollen, dass Ihre neue Klasse ein Frame

sein soll. Was ist aber, wenn Ihre neue Klasse ein Frame und ein auf das Klicken einer Schaltfläche achtendes Objekt sein soll? Kann Ihre neue Klasse beides sein?

Ja, das ist möglich. In Java gibt es etwas, das sich *Interface* nennt. Ein Interface ist wie eine Klasse, aber nicht damit identisch. Wenn Sie das Wort *Interface* hören, denken Sie ruhig zunächst an eine Klasse. Dann erinnern Sie sich an folgende Informationen:

- ✓ **Eine Klasse kann nur eine Elternklasse erweitern, aber eine Klasse kann mehr als ein Interface implementieren.**

Wenn Sie zum Beispiel möchten, dass `GameFrame` sowohl auf Tastatureingaben als auch auf Mausklicks achtet, können Sie schreiben:

```
class GameFrame extends JFrame
    implements ActionListener, ItemListener
```

- ✓ **Die Methoden eines Interface haben keinen eigenen Körper.**

Hier eine Kopie des API-Codes für das Interface `ActionListener`:

```
package java.awt.event;
import java.util.EventListener;
public interface ActionListener
    extends EventListener {
    public void actionPerformed(ActionEvent e);
}
```

Ich habe aus diesem Code die Kommentare entfernt und ansonsten alles so belassen, wie es im Prinzip im Original vorhanden ist. In diesem Code hat die Methode `actionPerformed` keinen Körper – keine geschweiften Klammern und keine Befehle, die ausgeführt werden sollen. Anstelle eines Körpers gibt es nur ein Semikolon.



Eine Methode ohne Körper wie die Methode, die im Interface `ActionListener` definiert wird, ist eine *abstrakte* Methode.

✓ Wenn Sie ein Interface implementieren, stellen Sie für alle Methoden des Interface Körper zur Verfügung.

Das ist der Grund dafür, dass in [Listing 15.1](#) die Methode `actionPerformed` erscheint. Indem der Code ankündigt, dass er das Interface `ActionListener` implementieren wird, ist er damit einverstanden, dass die Methode `actionPerformed` einen Sinn bekommt. Hier bedeutet dies, dass eine Methode `actionPerformed` deklariert wird, die über geschweifte Klammern, einen Körper und eventuell sogar über Befehle verfügt.

Wenn Sie ankündigen, dass Sie vorhaben, ein Interface zu implementieren, nimmt der Java-Compiler diese Ankündigung ernst. Wenn Sie dann später im Code vergessen, den Methoden des Interface einen Sinn zu geben, beschwert sich der Compiler massiv bei Ihnen.



Wenn Sie absolut keine Lust haben, selbst zu suchen, können Sie auf die Schnelle herausfinden, welche Methoden im Code zur Implementierung von Interfaces fehlen. Versuchen Sie, den Code zu kompilieren, und der Compiler listet alle Methoden auf, die Sie hätten deklarieren müssen.



[Kapitel 8](#) stellt `@Override` vor – eine Java-Annotation.

Normalerweise verwenden Sie `@Override`, um anzukündigen, dass eine Methode ersetzt wird, die bereits in einer Superklasse deklariert worden ist. Aber Sie können ab Java 6 `@Override` auch verwenden, um das Implementieren der Methode eines Interface zu signalisieren. Das ist genau das, was ich in [Listing 15.1](#) mache.

Threads

Hier ein wohlgehütetes Geheimnis: Java-Programme sind *multithreaded*. Dies bedeutet, dass mehrere Komponenten gleichzeitig ablaufen, wenn Sie ein Java-Programm ausführen. Der Computer führt den Code aus, den Sie geschrieben haben,

und er führt gleichzeitig noch anderen Code aus (Code, den Sie nicht geschrieben haben und den Sie auch nicht sehen). Während sich der Computer um die Befehle Ihrer Methode `main` kümmert, nimmt er sich eine Auszeit, schaut sich ein wenig um und führt Befehle anderer, unsichtbarer Methoden aus. Bei den meisten einfachen Java-Programmen handelt es sich bei diesen Java-Methoden um solche, die Teil der Java Virtual Machine (JVM) sind.

So gibt es zum Beispiel in Java einen Thread für die Ereignisbehandlung. Während Ihr Code ausgeführt wird, läuft der Code des Ereignisbehandlungsthreads im Hintergrund. Er achtet auf Mausklicks und unternimmt im Falle eines Klickens die entsprechenden Aktionen. [Abbildung 15.3](#) illustriert, wie das vonstattengeht.

Der Thread Ihres Codes	Der für die Ereignisbehandlung zuständige Thread
<code>setLayout(new FlowLayout());</code> <code>add(textField);</code> <code>add(button);</code> <code>add(label);</code> <code>button.addActionListener(this);</code> <code>pack();</code> <code>setVisible(true);</code>	Hat der Benutzer geklickt? . . Hat der Benutzer geklickt? . . Hat der Benutzer geklickt? Ja? Dann rufe ich die Methode <code>actionPerformed</code> auf.

[Abbildung 15.3](#): Java-Threads

Wenn der Benutzer die Schaltfläche anklickt, fragt der für die Ereignisbehandlung zuständige Thread: »Okay, die Schaltfläche ist angeklickt worden. Was soll ich nun tun?« Die Antwort lautet: »Rufe `actionPerformed`-Methoden auf.« Das ist so, als wenn der Thread Code wie diesen hier zur Verfügung hätte:

```
if (buttonJustGotClicked()) {
    object1.actionPerformed(infoAboutTheClick);
}
```

```
object2.actionPerformed(infoAboutTheClick);  
object3.actionPerformed(infoAboutTheClick);  
}
```

Klar, dass jede Antwort wieder zu einer Frage führt. In unserem Fall lautet die Anschlussfrage: »Wo findet der für die Ereignisbehandlung zuständige Thread `actionPerformed`-Methoden, die er aufrufen kann?« Und dann gibt es da noch die Frage: »Was ist, wenn Sie gar nicht wollen, dass der Thread die entsprechenden Methoden aufruft, die sich irgendwo im Code herumtreiben?«

Das ist der Moment, an dem die Methode `addActionListener` ins Spiel kommt. In [Listing 15.1](#) weist der Aufruf

```
button.addActionListener(this);
```

den für die Ereignisbehandlung zuständigen Thread an, die Methode `actionPerformed` des Codes auf die Liste mit den Methoden zu setzen, die aufgerufen werden sollen, und die Methode `actionPerformed` jedes Mal aufzurufen, wenn die Schaltfläche angeklickt wird.

Nun wissen Sie, wie das abläuft. Damit der Computer eine `actionPerformed`-Methode aufruft, registrieren Sie die Methode für den Java-Thread, der für die Ereignisbehandlung zuständig ist. Sie erledigen dies, indem Sie `addActionListener` aufrufen. Diese Methode gehört zu dem Objekt, auf dessen Angeklicktwerden (und andere Ereignisse) Sie warten. In [Listing 15.1](#) warten Sie darauf, dass das Objekt `button` angeklickt wird; also gehört die Methode `addActionListener` zu diesem Objekt.

Das Schlüsselwort »this«

In den [Kapiteln 9](#) und [10](#) gewährt Ihnen der Code in einer Methode über das Schlüsselwort `this` Zugriff auf Instanzvariablen. Da taucht die berechtigte Frage auf, was `this` nun wirklich bedeutet. Vergleichen Sie dieses Schlüsselwort mit dem deutschen Ausdruck »Ihr Name«.

Ich, (Ihr Name), erkläre hiermit feierlich, dass ich mein Amt als Schöffe ...

Der Ausdruck »Ihr Name« ist ein Platzhalter. Er ist ein Leerraum, in den eine Person ihren Namen einsetzen muss.

Ich, Paula Glück, erkläre hiermit ...

Ich, Gerd Johns, erkläre hiermit ...

Stellen Sie sich das Versprechen (*Ich ... erkläre hiermit feierlich, dass ich mein Amt ...*) als ein Stück Code in einer Java-Klasse vor. In diesem Code lautet der Ausdruck für den Platzhalter »Ihr Name«. Immer wenn eine Instanz der Klasse (eine Person) den Code ausführt (das Versprechen abgibt), ersetzt die Instanz den Platzhalter »Ihr Name« durch ihren eigenen Namen.

Das Schlüsselwort `this` arbeitet auf die gleiche Weise. Es befindet sich in dem Code, der die Klasse `GameFrame` definiert. Immer wenn eine Instanz von `GameFrame` aufgebaut wird, ruft die Instanz `addActionListener(this)` auf. In diesem Aufruf steht das Schlüsselwort `this` für die Instanz selbst.

```
button.addActionListener(dieseInstanzVonGameFrame);
```

Dadurch, dass die `GameFrame`-Instanz

`button.addActionListener(this)` aufruft, sagt sie: »Füge der Liste mit den Methoden, die aufgerufen werden, wenn die Schaltfläche angeklickt wird, meine Methode `actionPerformed` hinzu.« Da die `GameFrame`-Instanz über eine Methode `actionPerformed` verfügt, weil die Klasse `GameFrame` das `ActionListenerInterface` implementiert, passt alles wunderbar zusammen.

Das Innere der Methode

»actionPerformed«

Die Methode `actionPerformed` verwendet in [Listing 15.1](#) einige Tricks der Java-API, die die folgende Liste kurz zusammenfasst:

- ✓ Jede Instanz von `JTextField` (und `JLabel`) hat ihre eigenen Getter- und Setter-Methoden, zu denen auch `getText` und

`setText` gehören. Der Aufruf von `getText` schnappt sich alle Zeichen, die es in der Komponente gibt. In [Listing 15.1](#) entnimmt der umsichtige Einsatz von `getText` und `setText` dem Textfeld eine Zahl, die entweder durch nichts (dem leeren String `""`) oder durch die Zahl und das Wort `Ja!` ersetzt wird.

- ✓ Jede Komponente im Paket `javax.swing` (`JTextField`, `JButton` oder was es sonst noch gibt) verfügt über eine Methode `setEnabled`. Wenn Sie `setEnabled(false)` aufrufen, wird die Komponente ausgegraut, wird zu einem deaktivierten Objekt und ist nicht länger in der Lage, Klickereignisse der Schaltfläche zu empfangen.
- ✓ Jede Komponente im Paket `javax.swing` verfügt über eine Methode `requestFocus`. Wenn Sie `requestFocus` aufrufen, ist die Komponente in der Lage, wieder Eingaben des Benutzers anzunehmen. So besagt zum Beispiel in [Listing 15.1](#) der Aufruf von `textField.requestFocus()`, dass selbst dann ein Cursor in das Textfeld gesetzt werden kann, wenn der Benutzer die Schaltfläche gerade erst angeklickt hat. Dies soll dem Benutzer die Möglichkeit geben, einen neuen Rateversuch zu starten.



Sie können einen Test durchführen, um sicherzustellen, dass es sich bei dem Objekt, auf das von der Variablen für die Schaltfläche aus verwiesen wird, auch wirklich um das Ding handelt, das angeklickt worden ist. Schreiben Sie einfach `if (e.getSource() == button)`. Wenn Ihr Code zwei Schaltflächen mit den Namen `button1` und `button2` enthält, können Sie herausfinden, welche Schaltfläche angeklickt worden ist. Schreiben Sie `if (e.getSource() == button1)` und `if (e.getSource() == button2)`.

Die serialVersionUID

Kapitel 9 stellt die Annotation `SuppressWarnings` vor, die verhindert, dass Sie es mit etwas zu tun bekommen, das `serialVersionUID` genannt wird. Bei einer

`serialVersionUID` handelt es sich um eine Zahl, die Java hilft, Versionskonflikte zu vermeiden, wenn Sie ein Objekt von einem Ort an einen anderen senden. Sie können zum Beispiel den Status Ihres `JFrame`-Objekts an den Bildschirm eines anderen Computers senden. Der andere Computer ist dann in der Lage, die Versionsnummer des Frames zu überprüfen, damit nicht plötzlich irgendwelche komischen Dinge stattfinden.

Ich gehe in [Kapitel 9](#) nur am Rande auf `serialVersionUID` ein, indem ich Java anweise, Warnungen über fehlende Versionsnummern zu ignorieren. Dies ändert sich aber in [Listing 15.1](#). Mein `JFrame`-Objekt erhält eine richtige `serialVersionUID`. Da es sich hier um meine erste Version von `GameFrame` handelt, bekommt das Spiel die Versionsnummer 1. (Tatsächlich hat dieses `GameFrame` die Nummer `1L` erhalten (was den `long`-Wert 1 bedeutet; siehe [Kapitel 4](#).)

Die Frage ist nun, wann Sie sich damit abplagen müssen, die `serialVersionUID` zu ändern. Wenn die Version mit der Nummer 1 in Ordnung ist, ist dann Version 2 besser? Die Antwort ist ein wenig kompliziert, aber generell gilt, dass Sie die `serialVersionUID` erst neu vergeben, wenn Sie inkompatible Änderungen am Code der Klasse vorgenommen haben. Unter »inkompatible Änderungen« verstehe ich Änderungen, die es dem Code des empfangenden Computers unmöglich machen, mit den neu erstellten Objekten umzugehen. (Wenn Sie mehr über die `serialVersionUID` und darüber wissen wollen, was eine inkompatible Änderung am Code ausmacht, besuchen Sie die englischsprachige Website docs.oracle.com/javase/7/docs/platform/serialization/spec/version.html#6678.)



Java kennt einige grafische Werkzeuge, die Ihnen dabei helfen, eine GUI zu entwerfen. Wenn Sie www.eclipse.org/windowbuilder besuchen, stoßen Sie auf (englischsprachige) Informationen zum

WindowBuilder for Eclipse. Sie können mit dem WindowBuilder aus einer Palette Komponenten auf einen Frame ziehen. (Zu diesen Komponenten gehören Schaltflächen, Textfelder und andere Nettigkeiten.) Sie können dann mit Ihrer Maus die einzelnen Komponenten verschieben und ihre Größen ändern. Während Sie den Frame grafisch gestalten, erstellt Eclipse automatisch den Code des Frames. Zu jeder Komponente auf dem Frame gehört ein kleines tabellarisches Arbeitsblatt mit ihren Eigenschaften. Sie können zum Beispiel den Text auf der Oberfläche einer Schaltfläche ändern, indem Sie auf ihrem Arbeitsblatt den Eintrag unter `TEXT` ändern. Wenn Sie mit der rechten Maustaste auf ein Bild einer Komponente klicken oder dort ein `[Strg]`-Klicken ausführen, erhalten Sie die Möglichkeit, zur Methode `actionPerformed` der Komponente zu springen. In dieser Methode fügen Sie Java-Code wie `button.setText("Du hast mich angeklickt!")` hinzu. Ein Werkzeug wie WindowBuilder sorgt dafür, dass das Entwerfen von GUIs schneller, natürlicher und intuitiver vonstattengeht.



Dieses Kapitel beschreibt Funktionen des Java-Frameworks *Swing*. Swing ist seit 1998 Javas primäres Framework für die Entwicklung von GUI-Anwendungen. Gegen Ende 2011 fügte Oracle dem Java-Kern ein neues Framework hinzu – *JavaFX*. JavaFX stellt einen reichhaltigeren Satz an Komponenten zur Verfügung als Swing. Wenn es um einfache Anwendungen geht, ist JavaFX häufig zu kompliziert in der Anwendung. Wenn Sie daran interessiert sind, mehr über JavaFX zu erfahren, sollten Sie die amerikanische Website dieses Buches unter allmycode.com/JavaForDummies besuchen. Sie können auf dieser (englischsprachigen) Website GUI-Versionen vieler Beispiele dieses Buches herunterladen. Einige dieser GUI-Versionen verwenden JavaFX.

Auf andere Dinge als auf das Anklicken von Schaltflächen reagieren

Wenn Sie wissen, wie Sie auf eine Ereignisart reagieren müssen, sollten Sie keine Schwierigkeiten haben, auch mit anderen Ereignissen umgehen zu können. Die [Listings 15.3](#) und [15.4](#) zeigen ein Fenster an, das amerikanische Dollar in britische Pfund umrechnet. Der Code in diesen Listings reagiert auf viele Ereignisse. [Abbildung 15.4](#), [Abbildung 15.5](#) und [Abbildung 15.6](#) zeigen, wie der Code arbeitet.

```
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.text.NumberFormat;
import java.util.Locale;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
class MoneyFrame extends JFrame implements
    KeyListener, ItemListener, MouseListene
r {
    private static final long serialVersionUID = 1L;

    JLabel fromCurrencyLabel = new JLabel(" ");
    JTextField textField = new JTextField(5);
    JLabel label = new JLabel(" ");
    JComboBox combo = new JComboBox();
    NumberFormat currencyUS =
        NumberFormat.getCurrencyInstance();
    NumberFormat currencyUK =
        NumberFormat.getCurrencyInstance(Locale.UK);
    public MoneyFrame() {
```

```

        setLayout(new FlowLayout());
        add(fromCurrencyLabel);
        add(textField);
        combo.addItem("US in UK");
        combo.addItem("UK in US");
        add(label);
        add(combo);
        textField.addKeyListener(this);
        combo.addItemListener(this);
        label.addMouseListener(this);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE
    );

    setSize(300, 100);
    setVisible(true);
}

void setTextOnLabel() {
    String amountString = "";
    String fromCurrency = "";

    try {
        double amount =
            Double.parseDouble(textField.getText(
));
        if(combo.getSelectedItem().equals("US in
UK"))
        {
            amountString = " = " +
                currencyUK.format(amount * 0.61
214);
            fromCurrency = "$";
        }
        if(combo.getSelectedItem().equals("UK in
US"))
        {
            amountString = " = " +
                currencyUS.format(amount * 1.63
361);
            fromCurrency = "\u00A3";
        }

    } catch (NumberFormatException e) {
    }
    label.setText(amountString);
    fromCurrencyLabel.setText(fromCurrency);
}

```

```

@Override
public void keyReleased(KeyEvent k) {
    setTextOnLabel();
}

@Override
public void keyPressed(KeyEvent k) {
}

@Override
public void keyTyped(KeyEvent k) {
}

@Override
public void itemStateChanged(ItemEvent i) {
    setTextOnLabel();
}

@Override
public void mouseEntered(MouseEvent m) {
    label.setForeground(Color.red);
}

@Override
public void mouseExited(MouseEvent m) {
    label.setForeground(Color.black);
}

@Override
public void mouseClicked(MouseEvent m) {
}

@Override
public void mousePressed(MouseEvent m) {
}

@Override
public void mouseReleased(MouseEvent m) {
}
}

```

[Listing 15.3](#): Den Währungsrechner anzeigen

```

public class ShowMoneyFrame {

    public static void main(String args[]) {
        new MoneyFrame();
    }
}

```

[Listing 15.4](#): Den Code in [Listing 15.3](#) aufrufen

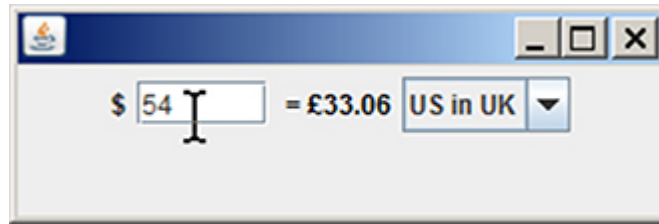


Abbildung 15.4: Währung umrechnen – Dollar in Pfund

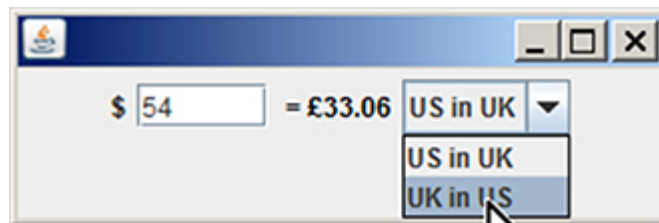


Abbildung 15.5: Das Listenfeld verwenden

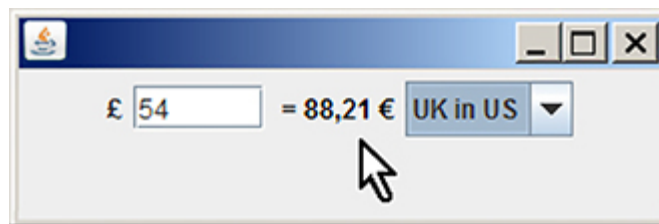


Abbildung 15.6: Währung umrechnen – Pfund in Dollar

Gut, [Listing 15.3](#) ist recht lang geworden, aber sein Code lässt sich gut zusammenfassen:

```
class MoneyFrame extends JFrame implements
   KeyListener, ItemListener, MouseListene
{
    Variablendeklarationen
    Konstruktor für die Klasse MoneyFrame
    Deklaration einer Methode setTextOnLabel
    Alle Methoden, die benötigt werden, weil die Klas
se
    drei Instanzen implementiert
}
```

Der Konstruktor fügt in [Listing 15.3](#) dem Fenster `MoneyFrame` die folgenden vier Komponenten hinzu:

- ✓ **Ein Label:** In [Abbildung 15.4](#) zeigt das Label ein Dollarzeichen an.
- ✓ **Ein Textfeld:** In [Abbildung 15.4](#) gibt der Benutzer 54 in das Textfeld ein.
- ✓ **Ein weiteres Label:** In [Abbildung 15.4](#) zeigt das zweite Label £33.06 an.
- ✓ **Ein Kombinationsfeld:** In [Abbildung 15.4](#) zeigt das Kombinationsfeld US in UK an. In [Abbildung 15.5](#) wählt der Benutzer im Feld ein Element aus. In [Abbildung 15.6](#) ist UK in US gewählt worden. (Ein Kombinationsfeld wird auch *Combobox* genannt.)

Der `MoneyFrame` implementiert drei Interfaces: `KeyListener`, `ItemListener` und `MouseListener`. Da drei Interfaces implementiert werden, kann der Code auf drei verschiedene Ereignisarten achten. Ich behandle die Interfaces und die Ereignisse in der folgenden Liste:

- ✓ **`KeyListener`:** Eine Klasse, die das Interface `KeyListener` implementiert, muss über die drei Methoden `keyReleased`, `keyPressed` und `keyTyped` verfügen. Wenn Sie Ihren Finger von einer Taste abheben, ruft der für die Ereignisbehandlung zuständige Thread `keyReleased` auf.

In [Listing 15.3](#) ruft die Methode `keyReleased` die Methode `setTextOnLabel` auf. Meine Methode `setTextOnLabel` prüft nach, was aktuell im Kombinationsfeld ausgewählt worden ist. Wenn der Benutzer die Option US in UK wählt, rechnet die Methode `setTextOnLabel` US-Dollar in englische Pfund um. Wenn der Benutzer die Option UK in US wählt, rechnet die Methode `setTextOnLabel` Pfund in Dollar um.



Ich verwende in der Methode `setTextOnLabel` den String `"\u00A3"`. Bei dieser ungewöhnlichen Zeichenfolge handelt es sich um das englische Währungssymbol Pfund. (Das `u` steht in `\u00A3` für *Unicode* – einem internationalen Standard für die Darstellung von Zeichen aus dem weltweiten Alphabet.) Wenn bei meinem Betriebssystem die Einstellungen für die Darstellung des Währungsformats standardmäßig das englische Währungssymbol anzeigen, erscheint das Pfundzeichen bei der Ausführung des Programms automatisch. Sie finden in Javas API-Dokumentation zur Klasse `Locale` mehr zu diesem Thema.

Nebenbei bemerkt, wenn Sie glauben, dass es sich hier um einen echten Währungsumrechner handelt, vergessen Sie's schnell. Dieses Programm verwendet Kurse, die stimmen können – oder auch nicht. Natürlich kann ein Programm auch das Internet erreichen, um tagesaktuelle Wechselkurse auszulesen, aber im Moment gibt es anderes, um das Sie sich im Zusammenhang mit Java kümmern müssen.

✓ **ItemListener:** Eine Klasse, die das Interface `ItemListener` implementiert, muss über die Methode `itemStateChanged` verfügen. Wenn Sie in einem Kombinationsfeld ein Element auswählen, ruft der für die Ereignisbehandlung zuständige Thread `itemStateChanged` auf.

Wenn der Benutzer im Kombinationsfeld `US in UK` oder `UK in US` wählt, ruft in [Listing 15.3](#) der für die Ereignisbehandlung zuständige Thread die Methode `itemStateChanged` auf. Im Gegenzug ruft die Methode `itemStateChanged` dann `setTextOnLabel` auf und so weiter.

✓ **MouseListener:** Eine Klasse, die das Interface `MouseListener` implementiert, muss über die Methoden `mouseEntered`, `mouseExited`, `mouseClicked`, `mousePressed` und `mouseReleased` verfügen. Die

Implementierung von `MouseListener` unterscheidet sich von der Implementierung von `ActionListener`. Wenn Sie `ActionListener` so implementieren, wie es [Listing 15.1](#) vormacht, reagiert der für die Ereignisbehandlung zuständige Thread nur auf Mausklicks. Aber durch `MouseListener` ist der Thread in der Lage, auch darauf zu reagieren, wenn der Benutzer die Maus drückt, loslässt oder bewegt und so weiter.

In [Listing 15.3](#) werden die Methoden `mouseEntered` und `mouseExited` aufgerufen, wenn Sie die Maus auf das Label oder davon wegbewegen. Woher wissen Sie, dass das Label in diese Aktion eingebunden wird? Schauen Sie sich einmal den Code im Konstruktor `MoneyFrame` an. Dort wird nämlich die Methode `addMouseListener` der Variablen `label` aufgerufen.

Werfen Sie in [Listing 15.3](#) auch einen kurzen Blick auf die Methoden `mouseEntered` und `mouseExited`. Wenn eine dieser beiden Methoden aufgerufen wird, macht der Computer weiter und ruft `setForeground` auf. Diese Methode ändert die Farbe des Labeltextes.

Ist das moderne Leben nicht wunderbar? In der Java-API gibt es sogar eine Klasse `Color` (deutsch *Farbe*) mit Namen wie `Color.red` und `Color.black`.

[Listing 15.3](#) enthält mehrere Methoden, die nicht wirklich verwendet werden. Wenn Sie beispielsweise `MouseListener` implementieren, muss Ihr Code eine eigene Methode `mouseReleased` enthalten. Sie benötigen diese Methode nicht, weil Sie irgendetwas machen wollen, wenn der Benutzer die Taste(n) seine Maus loslässt, sondern weil Sie dem Java-Compiler etwas versprochen haben und dieses Versprechen halten müssen.

Innere Klassen erstellen

Hier kommen tolle Neuigkeiten! Sie können eine Klasse in einer anderen Klasse definieren. Für den Benutzer verhält sich [Listing](#)

[15.5](#) so wie [Listing 15.1](#), aber in [Listing 15.5](#) gibt es in der Klasse `GameFrame` eine Klasse mit dem Namen `MyActionListener`.

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
class GameFrame extends JFrame {
    private static final long serialVersionUID = 1L;
    int randomNumber = new Random().nextInt(10) + 1;
    int numGuesses = 0;
    JTextField textField = new JTextField(5);
    JButton button = new JButton("Raten Sie");
    JLabel label = new JLabel(numGuesses + " Versuche"
);
    public GameFrame() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        add(textField);
        add(button);
        add(label);
        button.addActionListener(new MyActionListener())
);
        pack();
        setVisible(true);
    }
    class MyActionListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            String textFieldText = textField.getText();
            if (Integer.parseInt
                (textFieldText) == randomNumber) {
                button.setEnabled(false);
                textField.setText
                    (textField.getText() + " Ja!");
                textField.setEnabled(false);
            } else {
                textField.setText("");
                textField.requestFocus();
            }
            numGuesses++;
        }
    }
}
```

```

        String guessWord =
            (numGuesses == 1) ? " Versuch" : " Versu
che";
        label.setText(numGuesses + guessWord);
    }
}

```

[Listing 15.5](#): Eine Klasse in einer Klasse

Die Klasse `MyActionListener` ist in [Listing 15.5](#) eine *innere Klasse*. Eine innere Klasse ähnelt in vielen Dingen einer beliebigen anderen Klasse. Aber Sie sind in der Lage, im Code einer inneren Klasse auf die Felder der einschließenden Klasse zu verweisen. So verwenden zum Beispiel mehrere Befehle innerhalb von `MyActionListener` den Namen `textField`, und `textField` wird in der einschließenden Klasse `GameFrame` definiert.

Beachten Sie, dass der Code in [Listing 15.5](#) die Klasse `MyActionListener` nur einmal verwendet (für einen Aufruf von `button.addActionListener`). Deshalb frage ich, ob Sie wirklich einen Namen für etwas benötigen, das nur einmal verwendet wird? Das ist natürlich nicht der Fall. Sie können die gesamte Definition der inneren Klasse im Aufruf von `button.addActionListener` ersetzen. Wenn Sie das machen, erhalten Sie eine *anonyme innere Klasse*. [Listing 15.6](#) zeigt Ihnen, wie so etwas gemacht wird.

```

import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
class GameFrame extends JFrame {
    private static final long serialVersionUID = 1L;
    int randomNumber = new Random().nextInt(10) + 1;
    int numGuesses = 0;
    JTextField textField = new JTextField(5);
}

```

```

        JButton button = new JButton("Raten Sie");
        JLabel label = new JLabel(numGuesses + " Versuche"
    );
    public GameFrame() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        add(textField);
        add(button);
        add(label);
        button.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                String textFieldText = textField.getText(
    );

                if (Integer.parseInt
                    (textFieldText) == randomNumber)
{
                    button.setEnabled(false);
                    textField.setText
                        (textField.getText() + " Ja!");
                    textField.setEnabled(false);
                } else {
                    textField.setText("");
                    textField.requestFocus();
                }
                numGuesses++;
                String guessWord =
                    (numGuesses == 1) ?
                        " Versuch" : " Versuche";
                label.setText(numGuesses + guessWord);
            }
        });
        pack();
        setVisible(true);
    }
    class MyActionListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            String textFieldText = textField.getText();
            if (Integer.parseInt
                (textFieldText) == randomNumber) {
                button.setEnabled(false);
                textField.setText
                    (textField.getText() + " Ja!");
            }
        }
    }
}

```

```

        textField.setEnabled(false);
    } else {
        textField.setText("");
        textField.requestFocus();
    }
    numGuesses++;
    String guessWord =
        (numGuesses == 1) ? " Versuch" : " guess
es";
    label.setText(numGuesses + guessWord);
}
}
}

```

[Listing 15.6](#): Eine Klasse ohne Namen (innerhalb einer Klasse mit Namen)

Innere Klassen eignen sich gut für Ereignisbehandlungen wie die Methode `actionPerformed` in den Beispielen dieses Kapitels. Die meisten Probleme bereiten bei *anonymen* inneren Klassen die Anführungszeichen, die geschweiften Klammern und die Einrückungen – Sie dürfen sie nicht aus den Augen verlieren. Deswegen lautet mein bescheidener Rat, dass Sie für den Anfang Code ohne innere Klassen schreiben – Code wie den in [Listing 15.1](#). Wenn Sie sich später mit normalen Java-Klassen langweilen, probieren Sie es aus, einige dieser Klassen in innere Klassen umzuwandeln.

16

Java-Applets schreiben

In diesem Kapitel ...

- ▶ Ein einfaches Applet erstellen
 - ▶ Animationen für Applets entwerfen
 - ▶ Schaltflächen (und andere Dinge dieser Art) auf Applets unterbringen
-

Seit Java 1995 die Bühne stürmte, war das, was die Sprache so beliebt machte, das Konzept der Applets. Ein *Applet*, auf Deutsch *Miniaturanwendung* ist ein Java-Programm, das Bestandteil des Fensters eines Webbrowsers ist. Das Applet verfügt auf der Webseite über einen eigenen rechteckigen Bereich. Es ist in der Lage, eine Zeichnung anzuzeigen, ein Bild wiederzugeben, eine Figur in Bewegungen zu versetzen, auf Dateieingaben des Benutzers zu reagieren und alle möglichen interessanten Sachen zu unternehmen. Wenn Sie auf einer Webseite ein echtes, funktionsfähiges Programm unterbringen, verschaffen Sie sich ungeahnte Möglichkeiten.

Applets-Einmaleins

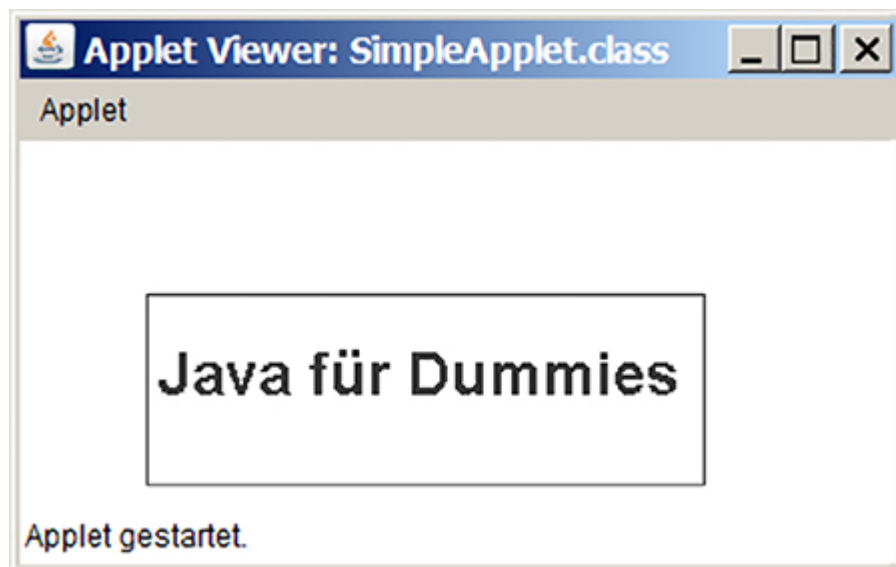
Die [Listings 16.1](#) und [16.2](#) enthalten ein sehr einfaches Java-Applet, das in einem rechteckigen Rahmen die Wörter Java für Dummies anzeigt (siehe [Abbildung 16.1](#)).

```
import javax.swing.JApplet;
public class SimpleApplet extends JApplet {
    private static final long serialVersionUID = 1L;
    public void init() {
        setContentPane(new DummiesPanel());
    }
}
```

[Listing 16.1](#): Ein Applet

```
import javax.swing.JPanel;
import java.awt.Font;
import java.awt.Graphics;
class DummiesPanel extends JPanel {
    private static final long serialVersionUID = 1L;
    public void paint(Graphics myGraphics) {
        myGraphics.drawRect(50, 60, 220, 75);
        myGraphics.setFont
            (new Font("Dialog", Font.BOLD, 24
));
        myGraphics.drawString("Java für Dummies", 55,
100);
    }
}
```

[Listing 16.2](#): Helfercode für das Applet



[Abbildung 16.1](#): Ein guter Titel!

Wenn Sie den Code der [Listings 16.1](#) und [16.2](#) ablaufen lassen, führen Sie keine Methode `main` aus. Stattdessen starten Sie einen Webbrowser, der dann wiederum auf eine HTML-Datei zugreift. Die HTML-Datei enthält einen Verweis auf den Java-

Code des Applets, und das Applet erscheint auf Ihrer Webseite. [Listing 16.3](#) zeigt, wie eine winzig kleine HTML-Datei aussieht.

```
<applet code=SimpleApplet width=350 height=200>  
</applet>
```

[Listing 16.3](#): Eine aus einer Zeile bestehende Webseite

Ich habe in [Abbildung 16.1](#) weder Firefox noch Internet Explorer oder einen anderen der üblichen Webbrowser verwendet, sondern *Java Applet Viewer* ausgeführt. Bei diesem Viewer handelt es sich um eine kleine, browserähnliche Anwendung, die extra zum Testen von Java-Applets entwickelt worden ist.

Auf einen Aufruf warten

Wenn Sie sich den Code in den [Listings 16.1](#) und [16.2](#) anschauen, fällt Ihnen vielleicht eines auf: Ein Applet hat keine Methode `main`. Der Grund dafür: Ein Applet ist kein vollwertiges Programm. Bei einem Applet handelt es sich um eine Klasse, die Methoden enthält, und Ihr Webbrowser ruft diese Methoden (direkt oder indirekt) auf. Sehen Sie in [Listing 16.1](#) die Methode `init`? Der Browser ruft diese Methode auf, die dann wiederum in [Listing 16.2](#) die Methode `setContentPane` aufruft.

Werfen Sie nun einen Blick auf die Methode `paint` in Listing 16.2. Der Browser ruft diese Methode automatisch auf, die dann den Browser darüber informiert, wie Ihr Applet auf dem Bildschirm gezeichnet werden soll. (Weiter hinten in diesem Kapitel im Abschnitt *Die Methoden in einem Applet* gibt es eine Liste mit speziellen Methoden, die Ihr Browser bei Applets aufruft.)

Eine öffentliche Klasse

Beachten Sie, dass es sich in [Listing 16.1](#) bei `SimpleApplet` um eine öffentliche Klasse handelt. Wenn Sie ein Applet erstellen und seine Klasse nicht zu einer öffentlichen machen, erhalten Sie einen »Applet not initied«- oder einen »Loading Java Applet Failed«-Fehler. Um es noch einmal klar und deutlich zu sagen: Jede Klasse, die `JApplet` erweitert, muss öffentlich

sein. Wenn dies nicht der Fall ist, kann Ihr Webbrowser die Methoden der Klasse nicht aufrufen.



Sie können entweder standardmäßig (`default`) oder öffentlich (`public`) auf eine Klasse zugreifen. Auf eine Klasse kann Code mit Standardzugriff nur verweisen, wenn er sich in demselben Paket wie die Klasse befindet. Stellen Sie sich nun vor, dass Ihr Webbrowser versucht, Methoden aufzurufen, die in der Klasse Ihres Applets vergraben sind. Da sich der Webbrowser wohl kaum in demselben Paket wie Ihr Applet befindet (glauben Sie mir, das ist nie der Fall), muss das Applet öffentlich sein. Wenn das Applet nicht öffentlich ist, kann der Code des Webbrowsers (Code, der sich nicht in demselben Paket wie das Applet befindet) keine der Methoden des Applets aufrufen. (Sie finden in [Kapitel 14](#) mehr zum Standardzugriff und zum öffentlichen Zugriff.)

(Noch einmal) die Java-API

Der Code in [Listing 16.2](#) verwendet ein paar interessante Kniffe aus der Java-API. Hier kommen Informationen, die bisher im Buch noch nicht vorgestellt worden sind:

✓ `drawRect`: Zeichnet ein nicht gefülltes Rechteck

Schauen Sie sich den Aufruf von `drawRect` in [Listing 16.2](#) an. Dem Aufruf entsprechend liegt die linke obere Ecke des Rechtecks 50 Pixel nach unten und 60 Pixel nach rechts von der linken oberen Ecke des Fensterelements entfernt. Die rechte untere Ecke des Rechtecks befindet sich 220 Pixel nach rechts und 75 Pixel nach unten von der linken oberen Ecke des Fensterelements entfernt.

Ich wollte, dass das Rechteck die Wörter `Java für Dummies` umschließt. Um Werte für den Aufruf von `drawRect` zu finden, habe ich mit Versuch und Irrtum gearbeitet. Sie können aber auch das Programm herausfinden lassen, wie viele Pixel `Java für Dummies` mit Beschlag belegt. Dafür

benötigen Sie die Klasse `FontMetrics`. (Informationen zu dieser Klasse finden Sie in der Dokumentation der Java-API.)

- ✓ Die Klasse `Font`: Beschreibt die Merkmale einer Schriftart
[Listing 16.2](#) erstellt eine fette, 24 Punkt große Schrift vom Typ `Dialog`. Andere Schrifttypen sind `DialogInput`, `Monospaced`, `Serif` und `SansSerif`.
- ✓ `drawString`: Zeichnet eine Folge von Zeichen
[Listing 16.2](#) zeichnet auf der Oberfläche des Fensterelements den String `Java für Dummies`. Die linke untere Ecke des Strings befindet sich 55 Pixel rechts und 100 Pixel unterhalb der linken oberen Ecke des Fensterelements.

Die Dinge in Bewegung versetzen

Das Applet dieses Abschnitts ist cool, weil es animiert ist – Sie sehen, wie sich auf dem Bildschirm ein Zähler ändert. Wenn Sie sich den Code dieses Applets anschauen, könnten Sie meinen, dass er sehr kompliziert ist. Gut, irgendwie ist er das auch, denn wenn Sie Java einsetzen, um Animationen zu erstellen, kommt schon einiges zusammen. Aber auf der anderen Seite enthält der Code für dieses Applets auch viele Allgemeinplätze. Wenn Sie eine eigene Animation erstellen wollen, können Sie dazu fast den gesamten Code dieses Abschnitts verwenden. Damit Sie wissen, worum es mir geht, schauen Sie sich die [Listings 16.4](#) und [16.5](#) an.

```
import javax.swing.JApplet;
import javax.swing.Timer;
import java.awt.Color;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
public class Odometer extends JApplet
                                implements ActionListener
{
    private static final long serialVersionUID = 1L;
```

```

Timer timer;
public void init() {
    OdometerPanel panel = new OdometerPanel();
    panel.setBackground(Color.white);
    setContentPane(panel);
}
public void start() {

    if (timer == null) {
        timer = new Timer(100, this);
        timer.start();
    } else {
        timer.restart();
    }
}
public void stop() {
    if (timer != null) {
        timer.stop();
        timer = null;
    }
}
public void actionPerformed(ActionEvent e) {

    repaint();
}
}

```

[Listing 16.4](#): Ein Zähler

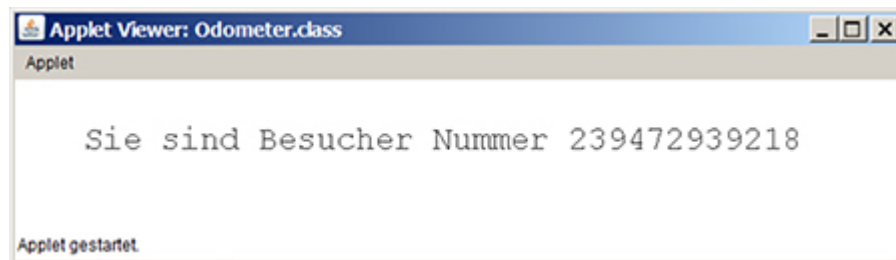
```

import javax.swing.JPanel;
import java.awt.Font;
import java.awt.Graphics;
class OdometerPanel extends JPanel {
    private static final long serialVersionUID = 1L;
    long hitCount = 239472938472L;
    public void paint(Graphics myGraphics) {
        myGraphics.setFont
            (new Font("Monospaced", Font.PLAIN, 24));
        myGraphics.drawString
            ("Sie sind Besucher Nummer " +
             Long.toString(hitCount++), 50, 50);
    }
}

```

[Listing 16.5](#): Das Fensterelement für den Zähler

[Abbildung 16.2](#) zeigt einen Schnappschuss von diesem Zähler (der im Original *Odometer* genannt wird). Beachten Sie in dieser Abbildung die Zahl. Sie ist nicht mit dem Startwert der Variablen `hitCount` identisch, da die App alle 100 Millisekunden den Wert von `hitCount` um 1 erhöht und den neuen Wert anzeigt. Der Zähler zeichnet keine echten Besucher auf, aber er ist nicht schlecht, oder?



[Abbildung 16.2](#): Eine beliebte Website

Die Methoden in einem Applet

Die meisten in den [Listings 16.4](#) und [16.5](#) verwendeten Methoden entsprechen dem Applet-Standard. Die Klassen `JApplet` und `JPanel` der Java-API enthalten für diese Methoden eine Standarddeklaration, weshalb Sie sich um eine solche Deklaration nicht kümmern müssen. Die einzigen Methoden, die Sie selbst in Ihrem Code unterbringen müssen, sind diejenigen, die Sie individuell erstellen wollen.

Hier eine Liste mit `JApplet`- und `JPanel`-Methoden, die Ihr Webbrowser automatisch aufruft:

- ✓ `init`: Der Browser ruft `init` auf, wenn Sie die Seite, die das Applet enthält, in einer Browsersitzung zum ersten Mal besuchen. Wenn Sie den Webbrowser dann schließen und die Seite mit dem Applet später erneut besuchen, ruft der Browser die Methode `init` des Applets erneut auf.
- ✓ `start`: Der Browser ruft `start` unmittelbar nach `init` auf. Wenn Ihr Applet eine fortlaufende Aufgabe ausführen soll, können Sie den Code dafür in der Methode `start` des Applets

beginnen. Wenn Ihr Applet zum Beispiel eine Animation enthält, gehört der Code, der die Ausführung der Animation startet, in die Methode `start`.

- ✓ `paint`: Der Browser ruft in [Listing 16.5](#) die Methode `paint` unmittelbar nach `start` auf. (Die Methoden `start` und `paint` befinden sich zwar in unterschiedlichen Klassen, aber der Browser kann herausfinden, wo das ist, weil in [Listing 16.4](#) `setContentPane` aufgerufen wird.) Die Methode `paint` enthält Anweisungen dafür, wie Ihr Applet auf den Bildschirm gezeichnet wird. Eine Erklärung hierzu finden Sie in [Kapitel 14](#).

Der Browser kann `paint` mehrmals aufrufen. Stellen Sie sich zum Beispiel vor, dass Sie einen Teil des Browserfensters mit einem anderen Fenster abdecken wollen. Oder Sie verkleinern den Browser so, dass nur noch ein Teil des Applets angezeigt wird. Wenn Sie dann später das Applet aufdecken oder das Browserfenster wieder vergrößern, ruft der Browser erneut die Methode `paint` des Fensterelements auf.

- ✓ `stop`: Wenn die Arbeit eines Applets abgeschlossen werden soll, ruft der Browser die Methode `stop` auf. Das ist so, als wenn Sie eine Verknüpfung anklicken, die Sie von der Seite mit dem Applet wegführt. Wenn Sie dann später die Seite mit dem Applet erneut besuchen, ruft der Browser wieder die Methode `start` des Applets auf.

Was alles in diese Methoden gehört

Der Code in den [Listings 16.4](#) und [16.5](#) verwendet ein Standardschema, um in einem Applet eine Animation zu erzeugen. Hier eine sehr kurze Erklärung dieses Schemas:

- ✓ Das Applet implementiert das Interface `ActionListener`.
- ✓ Die Methode `start` erstellt mit dem folgenden Code einen neuen Timer:

```
new Timer(100, this)
```

Der Timer in [Listing 16.4](#) schlägt alle 100 Millisekunden (jede Zehntelsekunde) Alarm.

In Wirklichkeit schlägt der Timer natürlich keinen Alarm, sondern er bringt Java dazu, eine `actionPerformed`-Methode aufzurufen. Hier stehen Sie wieder vor der Frage, wessen Methode das ist: Diese Frage beantwortet erneut das Schlüsselwort `this`. In [Listing 16.4](#) verweist es auf ein und denselben Code – auf diese Instanz des Objekts `Odometer`, das den Aufruf von `new Timer(100, this)` enthält. Also ruft Java in [Listing 16.4](#) alle 100 Millisekunden die Methode `actionPerformed` auf.

- ✓ Die Methode `actionPerformed` ruft die Methode `repaint` auf. Im Hintergrund sieht es dann so aus, dass ein Aufruf von `repaint` auch immer eine `paint`-Methode aufruft. In diesem Beispiel ist es die Methode `paint` im Code von Listing 16.5. Diese Methode `paint` zeichnet die Wörter *Sie sind Besucher Nummer wievielauchimmer* auf den Bildschirm.
- ✓ Irgendwann ist auch der längste Tag vorbei, und Ihr Browser ruft die Methode `stop` auf. Wenn dies geschieht, stopft diese Methode den Timer in den Müllcontainer.

Wenn es sich bei allem nicht um Standardcode handelte, hätte ich ein schlechtes Gewissen, alles nur so kurz zu erklären. Aber Sie können es mir glauben, dass Sie nur die [Listings 16.4](#) und [16.5](#) kopieren müssen, um in Ihrem Applet Bewegung hervorzubringen. Ersetzen Sie dann noch die dort vorhandenen Methoden `init` und `paint` durch eigenen Code.

Was könnten Sie in den Methoden `init` und `paint` unterbringen?

- ✓ **Wenn Sie eine Methode `init` deklarieren, sollte die Methode Installationscode für das Applet enthalten – den Kram, der abläuft, wenn das Applet zum ersten Mal geladen wird.**

In [Listing 16.4](#) hat der Installationscode mit einem Fensterelement zu tun:

- Er erzeugt ein Fensterelement, indem er den Konstruktor `OdometerPanel` aufruft.

- Er färbt den Hintergrund des Fensterelements weiß ein. (Dies sorgt dafür, dass sich das Rechteck im Applet gut vom Rest der Seite abhebt.)
- Er baut eine sehr enge Verbindung zwischen dem Fensterelement und dem Applet auf. Dies geschieht über den Aufruf der Methode `setContentPane`.

✓ Die Methode `paint` beschreibt eine Momentaufnahme der Bewegung des Applets.

Die Methode `paint` legt in [Listing 16.5](#) die Schriftart des Grafikpuffers fest, schreibt den Wert von `hitCount` auf den Bildschirm und erhöht dann `hitCount` um 1. (Wer braucht schon echte Besucher, wenn Sie Ihre Variable `hitCount` selbst erhöhen können?)

Die Variable `hitCount` beginnt mit einem großen Wert, der immer größer wird. Um so große Werte zu speichern, erhält `hitCount` den Typ `long`. Ich verwende die Methode `toString` der Klasse `Long`, um `hitCount` in einen String umzuwandeln. Sie können die Methode `toString` mit der Methode `parseInt` der Klasse `Integer` vergleichen (die ich in [Kapitel 11](#) vorstelle).



Um ein Applet zu debuggen, können Sie in seinem Code Aufrufe von `System.out.println` platzieren. Wenn Sie Eclipse verwenden, erfolgt die Ausgabe von `println` in der View CONSOLE.

Auf Ereignisse in einem Applet reagieren

In diesem Abschnitt gibt es ein interaktives Applet. Dieses Applet ist wie das Beispiel in [Kapitel 15](#). Ich habe sogar als Anfang von [Listing 16.7](#) Code aus [Listing 15.1](#) genommen. Das habe ich nicht aus Bequemlichkeit getan (obwohl jeder weiß, dass ich sehr faul

sein kann), sondern weil Applets sehr viel mit Frames gemein haben. Wenn Sie den Code für einen Frame nehmen und etwas »eindampfen«, können Sie normalerweise ein ordentliches Applet erstellen. Die [Listings 16.6](#) und [16.7](#) enthalten das Applet dieses Abschnitts.

```
import javax.swing.JApplet;
public class GameApplet extends JApplet {
    private static final long serialVersionUID = 1L;
    public void init() {
        setContentPane(new GamePanel());
    }
}
```

Listing 16.6: Ein Applet mit einem Ratespiel

```
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
class GamePanel extends JPanel implements ActionListener {
    private static final long serialVersionUID = 1L;

    int randomNumber = new Random().nextInt(10) + 1;
    int numGuesses = 0;
    JTextField textField = new JTextField(5);
    JButton button = new JButton("Raten Sie");
    JLabel label = new JLabel(numGuesses + " Versuche"
);
    GamePanel() {
        setBackground(Color.WHITE);
        add(textField);
        add(button);
        add(label);
        button.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e) {
        String textFieldText = textField.getText();
        if (Integer.parseInt(textFieldText)
```

```

er) {
    button.setEnabled(false);
    textField.setText
        (textField.getText() + " Ja!")
;
    textField.setEnabled(false);
} else {
    textField.setText("");
    textField.requestFocus();
}

numGuesses++;
String guessWord =
    (numGuesses == 1) ? " Versuch" : " Versuch
e";
    label.setText(numGuesses + guessWord);
}
}

```

Listing 16.7: Das Ratespiel

Um den Code der [Listings 16.6](#) und [16.7](#) ablaufen zu lassen, benötigen Sie eine HTML-Datei:

```

<applet code="GameApplet" width=225 height=50>
</applet>

```

[Abbildung 16.3](#) und [Abbildung 16.4](#) zeigen Ihnen, was geschieht, wenn Sie die Listings dieses Abschnitts ausführen. Das Ergebnis ist fast identisch mit dem, was der Code in [Listing 15.1](#) bewirkt. Der große Unterschied liegt darin, dass das Applet in einem Webbrowser als Teil einer Webseite erscheint.



Abbildung 16.3: Falsch geraten



Abbildung 16.4: Endlich richtig geraten

Statt sich auf den Code zu stürzen, den [Listing 16.7](#) enthält, sollten Sie sich um den Code kümmern, den es dort nicht gibt. Um aus [Listing 15.1](#) [Listing 16.7](#) zu machen, habe ich die folgenden Zeilen entfernt:

✓ **Ich plage mich nicht mehr mit dem Aufruf von `setLayout` ab.**

Das Standardlayout für ein Applet ist `FlowLayout`, das genau das ist, was mir vorschwebt. ([Kapitel 9](#) enthält Informationen über das `FlowLayout`.)

✓ **Ich rufe die Methode `pack` nicht auf.**

Die Felder `width` und `height` im HTML-Tag `applet` legen die Größe des Applets fest.

✓ **Ich rufe die Methode `setVisible` nicht auf.**

Ein Applet ist standardmäßig sichtbar.

Ein anderer auffälliger Unterschied existiert zwischen den [Listings 15.2](#) und [16.6](#). [Listing 16.6](#) verfügt – wie viele Applets – nicht über eine Methode `main`. Eine solche Methode wird hier nicht benötigt, weil Sie nirgendwo im Code `new GameApplet()` angeben müssen. Das erledigt der Webbrowser für Sie. Und nachdem der Webbrowser eine neue Instanz der Klasse `GameApplet` erstellt

hat, ruft er die Methode `init` der Instanz auf. Auf diese Weise werden Java-Applets standardmäßig ausgeführt.

17

Mit Java Datenbankverbindungen aufbauen und nutzen

In diesem Kapitel ...

- ▶ Mit einer Datenbank verbinden
 - ▶ Werte in eine Datenbank einfügen
 - ▶ Eine Datenbank abfragen
-

Immer wenn ich professionelle Java-Programmierer unterrichte, höre ich die gleiche Forderung: »Wir müssen keine niedlichen Zeichen über den Bildschirm fliegen lassen. Uns interessieren keine blinkenden Schaltflächen. Wir müssen auf Datenbanken zugreifen. Zeigen Sie uns einfach, wie wir ein Programm schreiben können, das sich mit Datenbanken unterhält.«

Deswegen kommt sie hier – Javas Datenbankverbindung.

JDBC und Java DB

Als ich anfang, mit Datenbanken zu arbeiten, war mein härtestes Problem, eine Verbindung zur Datenbank aufzubauen. Ich habe den gesamten Java-Code geschrieben. (Gut, ich habe diesen Code aus Büchern kopiert.) Der Java-Teil war einfach. Schwierig wurde es, als mein Code die Datenbank im System finden sollte.

Ein Teil des Problems lag darin, dass die Weise, wie Sie Ihren Code dazu bringen, sich mit der Datenbank zu unterhalten, von dem System abhängt, das Sie besitzen, und davon, was für eine Datenbank in Ihrem System läuft. Die Bücher, die ich damals

benutzt habe, konnten nicht auf alle Einzelheiten eingehen, weil sich die Computer der Leser zu stark voneinander unterscheiden und dadurch Probleme hervorriefen, die nichts mit Java zu tun hatten. Und nun schreibe ich selbst ein Kapitel über Datenbankverbindungen.

Glücklicherweise enthält das Java Development Kit (JDK) eine eigene Datenbank – *Java DB*. Java DB basiert auf der Datenbank *Apache Derby*. Java DB ist ein Leichtgewicht und basiert auf Standards. Java DB läuft problemlos mit dem Rest des Java JDK zusammen. Die Java-Gurus führten Java DB mit Java 6 ein.

Java DB erleichtert mein Leben, weil dadurch eine normale Datenbank zur Verfügung steht, die alle meine Leser nutzen können. Die Datenbank kostet nichts und muss nicht erst installiert werden.

Aber wie sieht es aus, wenn Sie Java DB nicht verwenden wollen? Was ist, wenn Ihre gesamten Daten in einer anderen Art von Datenbank abgelegt worden sind, zum Beispiel in MySQL, PostgreSQL, SQLite, Oracle, Microsoft Access, DB2 und so weiter? Auch dann hat Java eine Lösung für Sie! Die Klassen der Java Database Connectivity (JDBC) sorgen dafür, dass problemlos auf die üblichen Datenbankverwaltungssysteme zugegriffen werden kann. Besorgen Sie sich für das von Ihnen favorisierte System einen Treiber, passen Sie in den Beispielen dieses Kapitels zwei Zeilen an, und der Code sollte problemlos laufen.

Daten erstellen

Die Kernstücke von JDBC befinden sich in zwei Paketen: `java.sql` und `javax.sql`, die beide Bestandteil der Java-API sind. Die Beispiele dieses Kapitels, von denen [Listing 17.1](#) das erste ist, verwenden die Klassen in `java.sql`.

```
import java.sql.DriverManager;  
import java.sql.Statement;  
import java.sql.Connection;  
import java.sql.SQLException;
```

```

public class CreateTable {
    public static void main(String args[]) {
        final String DRIVER =
            "org.apache.derby.jdbc.EmbeddedDriver";
        final String CONNECTION =
            "jdbc:derby:AccountDatabase;create=true";
        try {
            Class.forName(DRIVER).newInstance();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        try (Connection connection =
            DriverManager.getConnection(CONNECTION)
;
            Statement statement =
                connection.createStatement()) {
            statement.executeUpdate(
                "create table KONTEN
"
                + " (NAME VARCHAR(32) NOT NULL PRIMARY KEY
"
                + "  ADRESSE VARCHAR(32),
"
                + "  SALDO FLOAT)
"
            );
            statement.executeUpdate(
                "insert into KONTEN values
                + " ('Barry Burd', '222 Cyber Lane
", 24.02)");
            statement.executeUpdate(
                "insert into KONTEN values
"
                + " ('Joe Dow', '111 Luddite Street
", 55.63)");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

[Listing 17.1](#): Eine Datenbank und eine Tabelle erstellen und Daten einfügen

Um anstelle von Java DB MySQL zu verwenden, nehmen Sie in [Listing 17.1](#) diese Änderungen vor: Ändern Sie den Wert von DRIVER in `"com.mysql.jdbc.Driver"`. Ändern Sie den Wert von CONNECTION in

`"jdbc:mysql://localhost/AccountDatabase;create=true"`. Nehmen Sie für DB2, Oracle und andere Datenbanken ähnliche Änderungen vor.

Um Datenbankcode auszuführen, müssen Sie über eine Datei verfügen, die einen passenden Datenbanktreiber enthält, und diese Datei muss sich an einem Ort befinden, an dem er von Java gefunden werden kann. Ich stelle in den Beispielen dieses Kapitels Verbindungen zu einer Java-DB-Datenbank her, die auch als Apache-Derby-Datenbank bekannt ist. Der Treiber befindet sich in einer Datei mit dem Namen `derby.jar`, die sich normalerweise im JDK-Verzeichnis `db/lib` befindet. Damit meine Java-Programme auf `db/lib/derby.jar` zugreifen können, habe ich diese `.jar`-Datei meinem Java-Classpath hinzugefügt.



Wie Sie eine `.jar`-Datei Ihrem Classpath hinzufügen, hängt von der IDE und möglicherweise auch von dem Betriebssystem ab, die Sie verwenden. In Eclipse wählen Sie PROJECT|PROPERTIES|JAVA BUILD PATH. Klicken Sie auf die Registerkarte LIBRARIES, dann auf die Schaltfläche ADD EXTERNAL JARs und navigieren Sie zum Verzeichnis `db/lib`. Bei anderen IDEs sehen diese Schritte etwas anders aus, ähneln sich aber im Prinzip.

Wenn Sie den Code in [Listing 17.1](#) ausführen, scheint nichts zu passieren. Das Programm startet und hält wieder an. Das ist alles. Der Code enthält keine sichtbare Ausgabe, weil alles, was der Code ausgibt, in einer Datenbank landet. Wenn Sie sehen wollen, welche Ergebnisse eine Ausführung des Codes in [Listing 17.1](#) liefert, müssen Sie nachsehen, welche Änderungen es an der Datenbank selbst gegeben hat. Lesen Sie also weiter!



Ich schreibe im vorherigen Absatz, dass es nur mäßig aufregend ist, den Code in [Listing 17.1](#) auszuführen, und ich schreibe, dass dabei nichts Sichtbares ausgegeben wird. Wenn Sie aber näher hinschauen, stoßen Sie doch auf einen Beleg dafür, dass der Code in [Listing 17.1](#) ausgeführt worden ist. Auf Ihrer Festplatte gibt es nun ein paar zusätzliche Dateien. Insbesondere die (neue) Datei `derby.log` enthält Text, der beschreibt, wann die Datenbanksoftware *Derby* gestartet und wann sie beendet wurde. Außerdem können Sie einen neuen Ordner mit dem Namen `derbyDB` finden, in dem es weitere Protokolldateien, eine Datei `service.properties` und einen Ordner mit `.dat`-Dateien gibt. (Diese Dateien enthalten das, was in der Datenbank gespeichert worden ist.) Wenn Sie Eclipse verwenden, können Sie diese neuen Ordner und Dateien sichtbar werden lassen, indem Sie im Package Explorer den Zweig Ihres Projekts anklicken und `FILE|REFRESH` wählen.

SQL-Befehle verwenden

Das Herz von [Listing 17.1](#) bilden drei Aufrufe von `executeUpdate`. Jeder dieser Aufrufe enthält einen String – einen normalen, in doppelten Anführungszeichen stehenden Java-String. Damit der Code lesbar bleibt, habe ich jeden String in seine Bestandteile zerlegt, die durch Pluszeichen (Javas Verknüpfungsoperator) separiert werden.

Javas Pluszeichen erfüllt hier zwei Aufgaben. Bei Zahlen führt es eine Addition durch. Bei Zeichenfolgen (Strings) fügt es zwei Strings zusammen und macht aus ihnen einen großen String.



Ein String, der von doppelten Anführungszeichen eingeschlossen wird, kann so lang werden, wie es Ihnen beliebt. Wenn Sie am rechten Bildschirmrand angekommen sind, schreiben Sie ruhig weiter. Wenn Sie den gesamten String sehen wollen, ohne scrollen zu müssen, zerlegen sie ihn so in einzelne Teile, wie ich das in [Listing 17.1](#) getan

haben. Grenzen Sie die Einzelteile dann durch Pluszeichen voneinander ab.



Sie können einen Java-String nicht dadurch in Einzelteile zerlegen, dass Sie die Taste `[Enter]` drücken und mit einer neuen Zeile anfangen. Wenn Sie einen String mit einem doppelten Anführungszeichen (") beginnen, muss sich das endende doppelte Anführungszeichen auf derselben Codezeile befinden.

Wenn Sie sich mit SQL (*Structured Query Language*; deutsch *strukturierte Abfragesprache*) auskennen, sagen Ihnen die Strings in den Aufrufen von `executeUpdate` etwas. Im anderen Fall verweise ich auf *SQL für Dummies* von Allen G. Taylor. Auf jeden Fall sollten Sie davon ausgehen, dass Sie in diesem Kapitel vergeblich nach Erklärungen für `create table` und `insert into` suchen, weil diese Befehle nichts mit Java zu tun haben. Aus der Sicht Javas handelt es sich dabei um Zeichenfolgen, die die Methode `executeUpdate` füttern. Diese Strings, die in SQL geschrieben worden sind, legen eine neue Datenbanktabelle an, der sie dann noch Datenzeilen hinzufügen. Wenn Sie ein Java-Datenbankprogramm schreiben, sind Aktionen wie die gerade beschriebenen Ihr tägliches Brot. Sie schreiben ganz normale SQL-Befehle und umgeben diese Befehle mit Aufrufen von Java-Methoden.



Der Code in diesem Kapitel hält sich ganz eng an die Techniken, die in JDBC 1.0 definiert werden. Spätere Versionen der JDBC-Klassen unterstützen etwas, das *scrollbare Ergebnismenge* (*Scrollable Result Sets*) genannt wird. Bei einer scrollbaren Ergebnismenge stehen Ihnen Methoden wie `insertRow` zur Verfügung, die es Ihnen ersparen, vollständige SQL-Befehlsfolgen schreiben zu müssen.

Verbinden und trennen

Der Code in [Listing 17.1](#) hat neben den Aufrufen der Methode `executeUpdate` auch noch sehr viel mit Kopieren und Einfügen zu tun. Hier ein Überblick über das, was die einzelnen Teile des Codes machen:

✓ **`Class.forName`:** Einen Datenbanktreiber suchen

Um mit einer Datenbank zu kommunizieren, benötigen Sie Software, die als eine Art Vermittler fungiert und *Datenbanktreiber* genannt wird. Treiber gibt es in allen Formen und Größen, und viele von ihnen kosten richtiges Geld. [Listing 17.1](#) verwendet einen kleinen, kostenlosen Treiber – den Derby JDBC Embedded Driver. Der Code für diesen Treiber befindet sich in der Klasse `EmbeddedDriver` (bei der es sich um eine Java-Klasse handelt). Diese Klasse ist Teil des Pakets `org.apache.derby.jdbc`.

Wenn Sie die Klasse `EmbeddedDriver` nutzen wollen, können Sie die Methode `Class.forName` aufrufen. Das ist kein Schreibfehler, aber die Java-API enthält tatsächlich eine Klasse mit dem Namen `Class`. In dieser Klasse stehen Informationen über Klassen, die der Java Virtual Machine (JVM) zur Verfügung stehen. In [Listing 17.1](#) hält der Aufruf von `Class.forName` nach der Klasse

`org.apache.derby.jdbc.EmbeddedDriver` Ausschau. Und nachdem dann eine Instanz von `EmbeddedDriver` geladen worden ist, können Sie mit dem Aufbau einer Verbindung zu einer Datenbank weitermachen.

✓ **`DriverManager.getConnection`:** Richtet eine Sitzung mit einer bestimmten Datenbank ein

Die Methode `getConnection` ist Bestandteil der Java-Klasse `DriverManager`. In [Listing 17.1](#) erstellt der Aufruf von `getConnection` eine `AccountDatabase` und öffnet eine Verbindung zu dieser Datenbank. Natürlich könnte die Datenbank `AccountDatabase` bereits vorhanden sein, bevor Sie den Code in [Listing 17.1](#) ausführen. In diesem Fall zeigt der String `;create=true` im Aufruf von `getConnection` keine Wirkung.

Beachten Sie bei den Parametern von `getConnection` (siehe [Listing 17.1](#)) die Doppelpunkte. Der Code gibt der Datenbank `AccountDatabase` nicht einfach nur ihren Namen, er informiert die Klasse `DriverManager` auch darüber, welche Protokolle für eine Verbindung mit der Datenbank verwendet werden sollen. Der Code `jdbc:derby:` – der dieselbe Aufgabe wie `http:` bei einer Webadresse erfüllt – weist den Computer an, das Protokoll `jdbc` zu verwenden, um mit dem Protokoll `derby` zu reden, das wiederum direkt mit Ihrer Datenbank `AccountDatabase` spricht.

- ✓ **`connection.createStatement`:** Erstellt einen Befehl
Das hört sich komisch an, aber Sie erstellen in der Java Database Connectivity ein einzelnes `statement`-Objekt. Nachdem Sie dies getan haben, können Sie das Objekt beliebig oft für viele verschiedene SQL-Strings verwenden, um viele verschiedene Befehle an die Datenbank zu übermitteln. Deshalb können Sie die Methode `statement.executeUpdate` erst aufrufen, nachdem Sie ein geeignetes `statement`-Objekt erstellt haben, was der Aufruf von `connection.createStatement` macht.
- ✓ **`try ... catch ...`:** Bestätigt Ausnahmen, die im Code geworfen werden können
Wenn Sie [Kapitel 13](#) gelesen haben, wissen Sie, dass Methodenaufrufe geprüfte Ausnahmen werfen. Bei einer *geprüften Ausnahme* handelt es sich um eine, die irgendwo im aufrufenden Code bestätigt wird. Ein Aufruf von `Class.forName` kann drei Arten von Ausnahmen werfen, und so gut wie alles andere in [Listing 17.1](#) ist in der Lage, eine `SQLException` zu werfen. Um die Ausnahmen zu bestätigen, erweitere ich meinen Code um `try-catch`-Befehle.
- ✓ **`try-mit-Ressourcen`:** Gibt Ressourcen frei – komme, was da wolle!

Jede Verbindung und jeder Datenbankbefehl kostet Systemressourcen. Wenn Sie mit Ihrer Arbeit fertig sind, sollten Sie diese Ressourcen wieder freigeben.

Das `try`-Befehl mit Ressourcen schließt im Anschluss an die Ausführung eines Befehls automatisch die Ressourcen und gibt sie wieder frei. Darüber hinaus kümmert sich diese Form des `try`-Befehls um den ganzen Kleinkram, der mit fehlgeschlagenen Versuchen zusammenhängt, Ausnahmen zu fangen. (Ich gehe in [Kapitel 13](#) ausführlicher auf `try-catch`-Befehle und auf `try`-Befehle mit Ressourcen ein.)

Daten abfragen

Was wollen Sie mit einer Datenbank anfangen, wenn sie Ihnen keine Daten liefert? Sie fragen in diesem Abschnitt die Datenbank ab, die Sie in [Listing 17.1](#) erstellt haben. [Listing 17.2](#) enthält den Code für eine solche Datenbankabfrage.

```
import static java.lang.System.out;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.ResultSet;
import java.text.NumberFormat;
public class GetData {
    public static void main(String args[]) {
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();
        final String DRIVER =
            "org.apache.derby.jdbc.EmbeddedDriver";
        final String CONNECTION =
            "jdbc:derby:AccountDatabase";
        try {
            Class.forName(DRIVER).newInstance();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

```

    }
    try (Connection connection =
        DriverManager.getConnection(CONNECTION);
        Statement statement =
            connection.createStatement();
        ResultSet resultset =
            statement.executeQuery(
                "select * from KONTEN"))
    {
        while(resultset.next()) {
            out.print(resultset.getString("NAME"));
            out.print(", ");
            out.print(resultset.getString("ADRESSE"));

;

            out.print(" ");
            out.println(currency.format(
                resultset.getFloat("SALDO")
            ));
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

Listing 17.2: Eine Datenbankabfrage



Um statt Java DB MySQL zu verwenden, bearbeiten Sie [Listing 17.2](#) wie folgt: Ändern Sie `DRIVER` in `"com.mysql.jdbc.Driver"`. Ändern Sie den Wert von `CONNECTION` in `"jdbc:mysql://localhost/AccountDatabase;create=true"`. Nehmen Sie für DB2, Oracle und andere Datenbanken ähnliche Änderungen vor.

Wenn Sie den Code in [Listing 17.2](#) ausführen, erhalten Sie das Ergebnis, das [Abbildung 17.1](#) zeigt. Der Code fragt die Datenbank ab, geht schrittweise durch deren Zeilen und gibt jede gefundene Zeile auf dem Bildschirm aus.

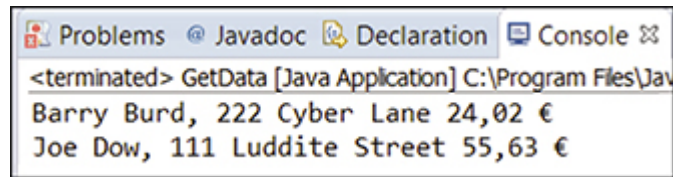


Abbildung 17.1: Von der Datenbank Daten erhalten

[Listing 17.2](#) beginnt mit den üblichen Aufrufen von `forName`, `getConnection` und `createStatement`. Dann ruft der Code `executeQuery` auf und übergibt dem Aufruf einen SQL-Befehl. Dieser Befehl holt sich alle Daten, die es in der Tabelle `KONTEN` (der Tabelle, die Sie in [Listing 17.1](#) anlegen) gibt.

Der Aufruf von `executeQuery` liefert etwas zurück, das vom Typ `java.sql.ResultSet` ist. (Das ist der Unterschied zwischen `executeUpdate` und `executeQuery`. `executeQuery` gibt eine Ergebnismenge – ein `ResultSet` – zurück, was `executeUpdate` nicht macht.) Die Ergebnismenge sieht wie eine Datenbanktabelle aus, die aus Zeilen und Spalten besteht. Jede Zeile enthält die Daten eines Kontos mit einem Namen, einer Adresse und einem Saldo.

Nachdem Sie `executeQuery` aufgerufen und Ihre Ergebnismenge erhalten haben, können Sie diese zeilenweise durchlaufen. Sie starten zu diesem Zweck eine kleine Schleife und testen zu Beginn jedes Durchlaufs die Bedingung `resultset.next()`. Dabei macht der Aufruf von `resultset.next()` jedes Mal Folgendes:

- ✓ Wenn eine weitere Zeile existiert, bringt er Sie zur nächsten Zeile der Ergebnismenge (dem nächsten Konto).
- ✓ Er informiert Sie darüber, dass eine weitere Zeile vorhanden ist, indem er einen `boolean`-Wert – `true` oder `false` – zurückgibt.

Wenn die Bedingung `resultset.next()` wahr ist, enthält die Ergebnismenge noch eine Zeile. Der Computer begibt sich zu dieser Zeile, und Sie marschieren zügig in den Körper der Schleife, um die Daten dieser Zeile auszulesen. Wenn aber

`resultSet.next()` falsch ist, gibt es in der Ergebnismenge keine weiteren Zeilen mehr. Sie verlassen die Schleife und beginnen damit, alles wieder zu schließen.

Stellen Sie sich nun vor, dass der Computer auf eine Zeile der Ergebnismenge zeigt und Sie sich noch in der Schleife aus [Listing 17.2](#) befinden. Sie erhalten von der Ergebnismenge die Daten der Zeile, indem Sie die Methoden `getString` und `getFloat` aufrufen. Sie haben in [Listing 17.1](#) die Tabelle `KONTEN` mit den Spalten `NAME`, `ADRESSE` und `SALDO` eingerichtet. In [Listing 17.2](#) holen Sie sich aus diesen Spalten Daten, indem Sie Ihre Methode *getDieseOderEineAndereDatenart* aufrufen und diesen Methoden die echten Spaltennamen übergeben. Nachdem Sie die Daten erhalten haben, zeigen Sie sie auf dem Bildschirm des Computers an.



Jede Instanz von `ResultSet` kennt in Java mehrere Methoden vom Typ `getDieseOderEineAndereDatenart`. Sie können – abhängig von Datentyp einer Spalte – Methoden wie `getArray`, `getBigDecimal`, `getBlob`, `getInt`, `getObject`, `getTimestamp` und so weiter aufrufen.

Teil VII

Der Top-Ten-Teil



Auf www.wiley-vch.de/publish/dt/books/ISBN3-527-71070-1
können Sie sich alle Codebeispiele des Buchs herunterladen.

In diesem Teil . . .

- ✓ Sie vermeiden allgemein übliche Fehler.
- ✓ Sie erkunden die besten Ressourcen für Java im Web.

18

Zehn Wege, um Fehler zu vermeiden

In diesem Kapitel ...

- ▶ Ihre Schreibweise und die Wertevergleiche überprüfen
 - ▶ Auf das Durchfallen achten
 - ▶ Methoden, Listeners und Konstruktoren dort unterbringen, wo sie hingehören
 - ▶ Statische und nicht statische Verweise verwenden
 - ▶ Andere schreckliche Fehler vermeiden
-

Die einzigen Menschen, die niemals Fehler machen, sind diejenigen, die nichts tun.« Das sagte einmal einer meiner Dozenten am College. Leider erinnere ich mich nicht mehr an seinen Namen und kann ihm nicht mehr meine Anerkennung zollen. Das ist mein Fehler.

Großbuchstaben dorthin setzen, wo sie hingehören

Java ist eine Sprache, die Groß-/Kleinschreibung beachtet. Sie müssen deshalb genau darauf achten, was Sie mit den einzelnen Buchstaben des Alphabets machen. Hier ein paar Dinge, die Sie sich merken sollten, wenn Sie Java-Programme schreiben:

- ✓ Java-Schlüsselwörter werden kleingeschrieben. So darf zum Beispiel in einem Java-Befehl das Wort *if* weder *If* noch *IF* sein.
- ✓ Wenn Sie Namen aus der Java-API verwenden, muss Ihre Schreibweise mit der in der API genau übereinstimmen.

✓ Außerdem müssen Sie dafür sorgen, dass Namen, die Sie selbst erfinden, im gesamten Programm auf ein und dieselbe Weise geschrieben werden. Wenn Sie eine Variable `meinKonto` deklarieren, können Sie nicht als `MeinKonto`, `meinkonto` oder `Meinkonto` darauf verweisen. Wenn Sie den Namen einer Variablen auf zwei verschiedene Arten schreiben, geht Java davon aus, dass Sie auf zwei verschiedene Variablen verweisen.

[Kapitel 3](#) enthält mehr Informationen zur Groß-/Kleinschrift unter Java.

Aus einer »switch«-Anweisung aussteigen

Wenn Sie nicht aus einer `switch`-Anweisung aussteigen, fallen Sie durch. Wenn zum Beispiel `verse` den Wert von 3 hat, gibt der folgende Code alle drei Zeilen aus – `Last refrain`, `He's a pain` und `Has no brain`.

```
switch (verse) {
case 3:
    out.print("Last refrain, ");
    out.println("last refrain,");
case 2:
    out.print("He's a pain, ");
    out.println("he's a pain,");
case 1:
    out.print("Has no brain, ");
    out.println("has no brain,");
}
```

[Kapitel 5](#) enthält die ganze Geschichte.

Werte mit einem doppelten Gleichheitszeichen vergleichen

Wenn Sie zwei Werte miteinander vergleichen, verwenden Sie ein doppeltes Gleichheitszeichen. Die Zeile

```
if (inputNumber == randomNumber)
```

ist richtig, aber die Zeile

```
if (inputNumber = randomNumber)
```

ist falsch. Warum das so ist, können Sie in allen Einzelheiten in [Kapitel 5](#) nachlesen.

Einer GUI Komponenten hinzufügen

Hier ein Konstruktor für einen Java-Frame:

```
public SimpleFrame() {  
    JButton button = new JButton("Danke ...");  
    setTitle("... Connie Santisteban und Brian Walls"  
);  
    setLayout(new FlowLayout());  
    add(button) ;  
    button.addActionListener(this);  
    setSize(300, 100);  
    setVisible(true);  
}
```

Was auch immer Sie machen, vergessen Sie auf keinen Fall den Aufruf der Methode `add`. Ohne diesen Aufruf können Sie zwar hart an einer Schaltfläche arbeiten, die dann aber niemals auf Ihrem Frame erscheint. [Kapitel 9](#) enthält eine Einführung in Themen dieser Art.

Listeners für Ereignisbehandlungen hinzufügen

Schauen Sie sich noch einmal den Code des vorherigen Abschnitts an, in dem ein `SimpleFrame` entworfen wird. Wenn

Sie den Aufruf von `addActionListener` vergessen, geschieht nichts, wenn Sie auf die Schaltfläche klicken. Es bringt auch nichts, die Schaltfläche ein zweites Mal noch stärker anzuklicken. [Kapitel 15](#) enthält einen Überblick über Listeners.

Die benötigten Konstruktoren definieren

Wenn Sie einen Konstruktor mit Parametern wie

```
public Temperature(double number)
```

definieren, erstellt der Computer keinen parameterlosen Standardkonstruktor mehr. Sie können also

```
Temperature raumTemp = new Temperature();
```

erst dann wieder aufrufen, wenn Sie explizit Ihren eigenen Konstruktor `Temperature` erstellt haben. Alle Einzelheiten hierzu finden Sie in [Kapitel 9](#).

Nicht statische Verweise reparieren

Wenn Sie versuchen, den folgenden Code zu kompilieren, erhalten Sie eine Fehlermeldung:

```
class WillNotWork {
    String greeting = "Hallo";
    public static void main(String args[]) {
        System.out.println(greeting);
    }
}
```

Zu dieser Fehlermeldung kommt es, weil `main` statisch, `greeting` aber nicht statisch ist. [Kapitel 10](#) enthält die vollständige Anleitung, wie Sie dieses Problem finden und beheben können.

Sich in den Grenzen eines Arrays aufhalten

Wenn Sie ein Array für zehn Komponenten definieren, erhalten diese Komponenten Indexe von 0 bis 9. Oder anders ausgedrückt: Wenn Sie

```
int guests[] = new int[10];
```

deklarieren, können Sie auf die Komponenten des Arrays `guests` verweisen, indem Sie `guests[0]`, `guests[1]` und so weiter bis `guests[9]` schreiben. Sie dürfen `guests[10]` nicht verwenden, weil das Array `guests` keine Komponente mit dem Index 10 enthält.

[Kapitel 11](#) enthält den neuesten Klatsch über Arrays.

»NullPointerException« vorhersehen

Die Beispiele in diesem Buch sind nicht für eine `NullPointerException` anfällig, aber Sie stoßen bei der alltäglichen Java-Programmierung ständig auf diese Ausnahme. Es kommt zu einer `NullPointerException`, wenn Sie eine Methode aufrufen, die zwar ein Objekt zurückgeben soll, dies aber nicht macht. Hier ein einfaches Beispiel:

```
import static java.lang.System.out;
import java.io.File;
class ListMyFiles {
    public static void main(String args[]) {
        File myFile = new File("\\windows");
        String dir[] = myFile.list();
        for (String fileName : dir) {
            out.println(fileName);
        }
    }
}
```

Dieses Programm zeigt eine Liste aller Dateien an, die sich im Verzeichnis `windows` befinden. (Wenn Sie wissen wollen, warum

es "\\windows" mit zwei umgekehrten Schrägstrichen heißen muss, siehe [Kapitel 8](#).)

Aber was geschieht, wenn Sie \\windows in etwas anderes ändern – in etwas, das keinen Namen eines Verzeichnisses darstellt?

```
File myFile = new File("&%$!!");
```

Dann gibt der Aufruf von `new File` den Wert `null` zurück (was in Java *nichts* bedeutet), wodurch die Variable `myFile` nichts enthält. Weiter hinten im Code verweist die Variable `dir` dann auf nichts, und der Versuch, die `dir`-Werte in einer Schleife zu durchlaufen, schlägt vollständig fehl. Dies beschert Ihnen eine tolle `NullPointerException`, und das Programm bricht mit lautem Getöse um Sie herum zusammen.

Um diese Art von Schwierigkeiten zu vermeiden, schauen Sie in der Dokumentation der Java-API nach. Wenn Sie eine Methode aufrufen könnten, die `null` zurückgeben könnte, erweitern Sie Ihr Programm um Code für eine Ausnahmebehandlung.

[Kapitel 13](#) erzählt die Geschichte über die Ausnahmebehandlung. Tipps darüber, wie Sie die API-Dokumentation lesen sollten, enthält [Kapitel 3](#).

Java bei der Suche nach seinen Dateien helfen

Sie kompilieren Java-Code und denken an nichts Böses, als der Computer einen `NoClassDefFoundError` anzeigt. Hier könnte zwar alles Mögliche schiefgelaufen sein, aber die Wahrscheinlichkeit ist hoch, dass der Computer eine Java-Datei nicht findet. Um dieses Problem zu beheben, müssen Sie Ordnung ins Durcheinander bringen.

- ✓ Alle Java-Dateien, deren Namen in Ihrem Code verwendet werden, müssen sich im Projektordner befinden.

✓ Wenn Sie benannte Pakete verwenden, muss Ihr Projektverzeichnis die dementsprechend benannten Unterverzeichnisse enthalten.

✓ Ihr Classpath muss sauber eingerichtet sein.

Detailliertere Hilfen zu diesem Thema finden Sie in [Kapitel 14](#).

Zehn Websites für Java

In diesem Kapitel ...

- ▶ Die Website (auch des Autors) zu diesem Buch
 - ▶ Ressourcen bei Oracle finden
-

Kein Wunder, dass das Web so beliebt ist. Es ist nicht nur nützlich, es macht auch Spaß. Dieses Kapitel beweist dies, indem es zehn nützliche und amüsante Websites aufführt. Jede dieser Sites enthält Ressourcen, die Ihnen dabei helfen, Java effektiver nutzen zu können. Und soweit ich weiß, verwendet keine dieser Sites AdWare, Popups oder andere unerfreuliche Sachen.

Und tun Sie sich selbst einen Gefallen. Nehmen Sie Ihre Lieblingssuchmaschine und stöbern Sie im Web nach deutschsprachigen Sites zu Java. Sie werden überrascht sein, wie viele es davon gibt.

Die Websites dieses Buches

Alle Beispiele dieses Buches können Sie unter www.wiley-vch.de/publish/dt/books/ISBN3-527-71070-1 herunterladen.

Wenn Sie auf zusätzliche technische Informationen und/oder die Originaldaten der Beispiele zugreifen möchten, besuchen Sie die englischsprachige Website des Autors unter www.allmycode.com/JavaForDummies.

Aus erster Hand

Oracles offizielle Website für Java ist
www.oracle.com/technetwork/java.

Endbenutzer der Java-Technologie sollten www.java.com
(deutschsprachig) besuchen.

Programmierer und Entwickler, die daran interessiert sind, sich
mit der Java-Technologie auseinanderzusetzen, gehen zu
www.java.net.

News, Berichte und Beispielcode

Artikel von Experten finden Sie bei InfoQ unter www.infoq.com.

Für Diskussionen mit vielen (auch sehr cleveren) Leuten
besuchen Sie JavaRanch unter www.javaranch.com.

Es gibt nicht nur Englisches

Die Java User Group Deutschland finden Sie unter
www.java.de.

Jedermanns Lieblingssite

Es ist wahr – diese beiden Sites sind nicht ausschließlich Java
gewidmet. Aber keine Liste für Computerfreaks wäre ohne
Slashdot und SourceForge vollständig.

Slashdots Slogan lautet: »News for nerds, stuff that matter.« Das
sagt eigentlich alles. Besuchen Sie <http://slashdot.org>.

Die SourceForge-Fundgrube (unter
<http://sourceforge.net>) beherbergt mehr als 200.000
kostenlose Open-Source-Projekte. Ein Besuch lohnt sich!



LAUSCHEN & LESEN

Stichwortverzeichnis

A

[actionPerformed](#)

Anführungszeichen

[doppelte](#)

[einfache](#)

[Animation](#)

[Annotation](#)

[SuppressWarnings](#)

[Anweisung](#)

Apache Derby [1](#), [2](#),

[API](#)

[Applet](#)

[Applet Viewer](#)

[Ereignisse](#)

[Installationscode](#)

[Standardmethoden](#)

Application Programming Interface

args[]

Argument

Befehlszeile

Array 1, 2,

ArrayList

deklarieren

Element

erstellen

for-Schleife

Index

Initialisierer

Komponente

Länge

Objekte

suchen

Werte speichern

zweidimensionales

ASCII

Assembly Line

[formulieren](#)

[Ausdruck](#)

[Ausnahme](#)

[bestätigen](#)

[erstellen](#)

[Exception Handling](#)

[fangen](#)

geprüfte [1](#), [2](#), [3](#),

[Schlüsselwörter](#)

[Typen](#)

ungeprüfte [1](#), [2](#),

[werfen](#)

[Ausnahmebehandlung](#)

B

[Bedingung](#)

[Befehl](#)

[verschachtelter](#)

[Zuweisungsbefehl](#)

Befehlszeile

[Argumente](#)

Befehlszeilenargument

[verwenden](#)

Bezeichner [1](#), [2](#),

[Bit](#)

[Block](#)

[Variablendeklaration](#)

[break](#)

[Bytecode](#)

[catch](#)

[multi-catch](#)

[passende Klausel](#)

[Verarbeitung](#)

C

[Classpath](#)

[Code](#)

[auskommentieren](#)

[einfügen](#)

[einrücken](#)

[kopieren](#)

[Codenamen](#)

[Combobox](#)

Compiler [1](#), [2](#), [3](#),

D

Datei

[auslesen](#)

[kopieren](#)

[schließen](#)

[schreiben in](#)

[suchen](#)

[Verbindung beenden](#)

[Dateibehandlung](#)

Daten

[einfügen](#)

[speichern in Datei](#)

[Datenbank](#)

[abfragen](#)

[erstellen](#)

Datenbanktreiber [1](#), [2](#),

[Datentyp](#)

[DB2](#)

[DecimalFormat](#)

[default](#)

[Deklaration](#)

[Variable](#)

Dezimalzeichen [1](#), [2](#),

Dialogfeld

[anzeigen](#)

[Zahleneingabe](#)

[Division](#)

do [1](#), [2](#),

[Dokumentation](#)

[Durchfallen](#)

E

Eclipse

[installieren](#)

[starten](#)

[WindowBuilder](#)

[Elternklasse](#)

[Entwicklungsversion](#)

[enum](#)

[EOF](#)

[Ereignis](#)

[Applet](#)

Ereignisbehandlung [1](#), [2](#),

[Ergebnis zurückgeben](#)

[Ergebnismenge, scrollbare](#)

[Escape-Folge](#)

[Exception](#)

[Exception](#) *siehe* [Ausnahme](#)

extends

F

Feld 1, 2,

geschütztes

sperren

Fenster

Schaltfläche

final

Array.

Formatierungsstring

Frame

erstellen

Größe

Zeichnung

G

Ganzzahl

auslesen

speichern

Generics

Getter

Methode

Gleichheitszeichen

doppeltes

Grafikpuffer

GUI 1, 2,

Komponenten hinzufügen

|

IDE 1, 2,

if, Befehl 1, 2,

Import

import, Deklaration

Importdeklaration

Initialisierer

Array

statischer

[Initialisierung](#)

[Instanz](#)

[Instanzenvariable](#)

[Integer, Klasse](#)

[Interface](#)

[Interpreter](#)

[Iteration](#)

[Iterator](#)

J

Java

[erste Schritte](#)

[Grammatik](#)

[herunterladen](#)

[Historie](#)

[SE API Docs](#)

[Java 8](#)

[Java DB](#)

[Java Development Kit](#)

[Java Runtime Environment](#)

[Java SE](#)

Java Virtual Machine [1](#), [2](#),

[java.sql](#)

[Javadoc-Kommentar](#)

Javadoc

[Programm](#)

[Webseiten erstellen](#)

[JavaFX](#)

[javax.sql](#)

[JButton](#)

[JDBC](#)

[JDK](#)

[JLabel](#)

[JRE](#)

JUnit

JVM

[anhalten](#)

K

Kindklasse

Klammer

geschweifte [1](#), [2](#),

Klasse [1](#), [2](#),

[anonyme innere](#)

[BigDecimal](#)

definieren [1](#), [2](#), [3](#),

[Double](#)

Elternklasse [1](#), [2](#),

[erweitern](#)

[Haus](#)

[innere](#)

[innere erstellen](#)

[Instanz](#)

Kindklasse [1](#), [2](#),

[nicht öffentliche](#)

öffentliche [1](#), [2](#), [3](#),

[public](#)

[Sammelklasse](#)

[Sammelklassen, Übersicht](#)

Superklasse [1](#), [2](#),

[Swing](#)

[Unterklasse](#)

[Unterklassen](#) *siehe* [Unterklasse](#)

[Zugriffsmodifizierer](#)

Klausel

[catch, Parameter](#)

[finally](#)

[throws](#)

[Kombinationsfeld](#)

[Kommentar](#)

[Javadoc](#)

Konstruktor [1](#), [2](#), [3](#),

[definieren](#)

[deklarieren](#)

[Standardkonstruktor](#)

[Unterklassen](#)

[Vererbung](#)

L

Lambda-Ausdruck

[Klassifizierung](#)

[verwenden](#)

[zwei Parameter](#)

Layout

[FlowLayout](#)

[GridLayout](#)

Listener

[ItemListener](#)

[KeyListener](#)

[MouseListener](#)

Literal

M

MapReduce

Methode

[abstrakte](#)

add 1, 2,

[aufrufen](#), [statische](#)

[Character.toUpperCase](#)

[close](#)

[definieren](#)

[delete](#)

[display](#)

[equals](#)

[Ergebnis zurückgeben](#)

[exists](#)

[Fabrikmethode](#)

[Factory Method](#)

[Get](#)

[hasNext](#)

[main](#)

[Objekt zurückgeben](#)

[Parameterliste](#)

[printf](#)

[remove](#)

[Set](#)

[setDefaultCloseOperation](#)

[setLayout](#)

[setSize](#)

[setTitle](#)

[setVisible](#)

[sleep](#)

[sum](#)

[überschreiben](#)

[Wert übergeben](#)

[Wert zurückgeben](#)

[Werte empfangen](#)

[Werte senden](#)

[Zugriffsmethoden](#)

[Methodenaufruf](#)

[Klassenname](#)

[platzieren](#)

[Methodendeklaration](#)

[Methodenkopf](#)

[Methodenkörper](#)

[Methodenreferenz](#)

[Miniaturanwendung](#) *siehe* [Applet](#)

[Minuszeichen](#)

[Mitglied](#)

[geschütztes](#)

[Zugriffsmodifizierer](#)

[Multicoreprozessor](#)

[nutzen](#)

[Multiplikation](#)

[Multithreaded](#)

[MySQL](#)

N

[Nachkommastelle](#)

[Name, vollqualifizierter](#)

[next](#)

[nextBoolean](#)

[nextDouble](#)

[nextInt](#)

[NoClassDefFoundError](#)

[null](#)

[NullPointerException](#)

O

Objekt

[Array](#)

[erstellen](#)

[Felder verwenden](#)

[vergleichen](#)

[zurückgeben](#)

OOP

[Terminologie](#)

Operator

[Bedingungsoperator](#)

[Dekrement](#)

Diamantoperator [1](#), [2](#),

[Inkrement](#)

[logische](#)

[Postdekrement](#)

[Prädekrement](#)

[Präinkrement](#)

[Rest-Operator](#)

[Vergleichsoperatoren](#)

[Zuweisungsoperatoren](#)

Oracle

Oval zeichnen

pack

P

Paket

Namensgebung

unbenanntes

Paketdeklaration

Parameter

übergeben

Parameterliste 1, 2, 3,

Pluszeichen 1, 2,

Portierbarkeit

Postinkrement

PrintStream

private

Produktversion

Programm

Ausführung anhalten

Gliederung

mehrere Ressourcen

Programmiersprache

Programmierung

funktionale

imperative

objektorientierte 1, 2,

Q

Quellcode

R

Rechteck zeichnen

Referenz übergeben

Referenzieren

Referenztyp 1, 2,

Regel erzwingen

Ressource, mehrere

return 1, 2,

Rückgabetyp

Rückgabewert

S

Sammelklasse

Schaltfläche

Schleife

do

for

while

zählende

Schleifendurchlauf

Schlüsselwort 1, 2,

Schriftart

Scrollable Result Set

Scrollbare Ergebnismenge

SDK

Seiteneffekt

Semikolon

serialVersionUID

Setter, Methode

Software

Software Development Kit

SQL

Befehle verwenden

Stacktrace

printStackTrace

Standardzugriff

static 1, 2,

Stream

Parallel

verwenden

String [1](#), [2](#),

[in switch-Befehlen](#)

[umwandeln in int-Wert](#)

[vergleichen](#)

[Suchen](#)

[super](#)

[Superklasse](#)

[Swing](#)

switch [1](#), [2](#),

[enum](#)

T

[Tabelle erstellen](#)

[Tastatureingabe](#)

Text

[anzeigen](#)

[zeilenweise lesen](#)

[Texteingabe auslesen](#)

Textfeld

this [1](#), [2](#), [3](#),

Thread

[try](#)

[mit Ressourcen](#)

Typ [1](#), [2](#),

generischer [1](#), [2](#),

U

Überschreiben

Unicode [1](#), [2](#),

Unterklasse

[definieren](#)

[erstellen](#)

[Konstruktoren](#)

[verwenden](#)

V

Variable

[deklarieren](#)

initialisieren [1](#), [2](#),

[Instanzvariable](#)

[Instanzvariable](#)

[Klassenvariable](#)

[methodenlokale](#)

[Nachkommastellen](#)

[platzieren](#)

[Terminologie](#)

Typ [1](#), [2](#),

[Wert](#)

Variablendeklaration [1](#), [2](#),

[kombinieren](#)

[Variablenname](#)

Variablentyp

[BigDecimal](#)

[boolean](#)

[char](#)

[Datentyp](#)

[double](#)

[float](#)

[ganzzahlige Typen](#)

[int](#)

[JFrame](#)

[primitiver](#)

[Referenztyp](#)

[String](#)

[Vererbung](#)

[Konstruktor](#)

[Verzeichnisname](#)

[Verzeichnisstruktur](#)

[void](#)

W

[Währungsbetrag anzeigen](#)

[Website, Überblick](#)

[Wert](#)

[speichern in Array](#)

übergeben [1](#), [2](#),

[vergleichen](#)

[zurückgeben](#)

[while](#)

Z

Zahl

formatieren [1](#), [2](#),

[Formatierung](#)

[runden](#)

[umwandeln](#)

[Zahlenformat, deutsch](#)

[Zeichen auslesen](#)

[Zeichenkodierung](#)

[Zeichnung, Frame](#)

[Zufallszahl](#)

[geschützter](#)

[Standardzugriff](#)

[Zugriffsmodifizierer](#)

Zugriffsmethode [1](#), [2](#), [3](#),

[Regeln erzwingen](#)

[Klasse](#)

[Mitglieder](#)