O'REILLY®

Aktualisierte Neuauflage des **Bestsellers**

Java

von Kopf bis Fuß

Eine abwechslungsreiche Entdeckungsreise durch die objektorientierte Programmierung

Kathy Sierra, Bert Bates & Trisha Gee

Übersetzung von Jørgen W. Lang

Ein gehirnfreundliches Buch

Inhalt

Cover

Titel

Impressum

Widmung

Die Autoren

Inhaltsverzeichnis

i Einführung

Für wen ist dieses Buch?

Wir wissen, was Sie denken.

Und wir wissen, was Ihr Gehirn gerade denkt.

Das haben WIR getan

Was Sie für dieses Buch brauchen

Die Fachgutachter der dritten englischen Auflage

Die Fachgutachter der zweiten englischen Auflage

1 Die Oberfläche durchbrechen

Wie Java funktioniert

Was Sie in Java tun werden

Eine sehr kurze Geschichte von Java

Codestruktur in Java

Eine Klasse mit einer main()-Methode schreiben

Einfache boolesche Tests

Bedingte Verzweigungen

Eine ernsthafte Geschäftsanwendung schreiben

Der Phras-O-Mat

Übungen

2 Die Reise nach Objectville

Stuhlkriege

Ihr erstes Objekt erstellen

Die Film-Objekte erstellen und testen

Schnell! Nichts wie raus aus main()!

Das Ratespiel ausführen

Übungen

3 Verstehen Sie Ihre Variablen

Eine Variable deklarieren

»Einen doppelten Espresso bitte, ach nein, doch lieber einen int.«

Finger weg von Schlüsselwörtern!

Ein Dog-Objekt kontrollieren

Eine Objektreferenz ist einfach ein anderer Variablenwert.

Das Leben auf dem Garbage Collectible Heap

Ein Array ist wie ein Schrank voller Tassen … ähm Becher

Ein Dog-Beispiel

Übungen

4 Wie sich Objekte verhalten

Erinnern Sie sich: Eine Klasse beschreibt, was ein Objekt weiß und was es tut.

Die Größe beeinflusst das Bellen

Sie können einer Methode Informationen schicken

Sie können von Methoden etwas zurückbekommen

Sie können einer Methode mehr als eine Sache übergeben

Parameter und Rückgabetypen

Kapselung

Wie verhalten sich Objekte in einem Array?

Instanzvariablen deklarieren und initialisieren

Variablen vergleichen (elementare und Referenztypen)
Übungen
5 Methoden mit Superkräften
Ein Spiel im Schiffe-versenken-Style: »Startups versenken«
Eine Klasse entwickeln
Die Methodenimplementierungen schreiben
Den Testcode für die SimpleStartup-Klasse schreiben
Die checkYourself()-Methode
Vorcode für die SimpleStartupGame-Klasse
Die main()-Methode des Spiels
Dann spielen wir mal …
Mehr über for-Schleifen
Die verbesserte for-Schleife
Elementare Typen casten
Übungen
6 Die Java-Bibliothek verwenden
Das vorige Kapitel endete mit einem Cliffhanger – nämlich einem Bug.
Wachen Sie endlich auf und sehen Sie der Java-API ins Auge
Einige Dinge, die Sie mit ArrayList tun können
ArrayList mit einem regulären Array vergleichen
Bauen wir das RICHTIGE Spiel: »Startups versenken«
Vorcode für die echte StartupBust-Klasse
Die finale Version der Startup-Klasse
Boolesche Ausdrücke mit Superkräften
Die Bibliothek (Java-API) verwenden
Übungen
7 Besser leben in Objectville
Stuhlkriege neu aufgelegt …

Vererbung verstehen

Entwerfen wir einen Vererbungsbaum für ein Tiersimulationsprogramm

Weitere Vererbungsmöglichkeiten finden

IST EIN und HAT EIN verwenden

Woher wissen Sie, dass Ihre Vererbungshierarchie korrekt ist?

Vererbung kann man gut gebrauchen, man kann sie aber auch missbrauchen!

Den Vertrag einhalten: Regeln für das Überschreiben

Eine Methode überladen

Übungen

8 Ernsthafte Polymorphie

Haben wir etwas vergessen, als wir das hier entworfen haben?

Der Compiler verhindert die Instanziierung von abstrakten Klassen

Abstrakt vs. konkret

Abstrakte Methoden MÜSSEN implementiert werden

Polymorphie in Aktion

Was ist mit Nicht-Animals? Warum machen wir die Klasse nicht so allgemein, dass sie mit allem umgehen kann?

Wenn ein Dog sich nicht wie ein Dog verhält

Erforschen wir ein paar Entwurfsoptionen

Das Pet-Interface erstellen und implementieren

Die Superklassenversion einer Methode aufrufen

Übungen

9 Leben und Sterben eines Objekts

Der Stack und der Heap: wo das Leben spielt

Methoden werden gestapelt

Was ist mit lokalen Variablen, die Objekte sind?

Das Wunder der Objekterstellung

Ein Duck-Objekt konstruieren

Erstellt der Compiler nicht immer einen argumentlosen Konstruktor für Sie?

Kurzer Rückblick. Vier Dinge, die Sie sich zu Konstruktoren merken sollten!

Die Rolle der Superklassenkonstruktoren im Leben eines Objekts

Kann ein Kind vor seinen Eltern existieren?

Was ist mit Referenzvariablen?

Mir gefällt nicht, worauf das hinausläuft.

Übungen

10 Zahlen, bitte!

MATH-Methoden: Näher werden Sie einer globalen Methode nie wieder kommen

Der Unterschied zwischen regulären (nicht statischen) und statischen Methoden

Eine statische Variable initialisieren

Math-Methoden

Elementartypen verpacken

Autoboxing funktioniert fast überall

Und umgekehrt … eine elementare Zahl in einen String verwandeln

Zahlenformatierung

Der Format-Spezifizierer

Übungen

11 Datenstrukturen

Die java.util-API, List und Collections entdecken

Generics sorgen für mehr Typsicherheit

Erneut ein Blick auf die sort()-Methode

Die neue, verbesserte Song-Klasse

Mit Comparators sortieren

Die Jukebox mit Lambdas aktualisieren

Ein HashSet anstelle einer ArrayList verwenden
Was Sie über TreeSet wissen MÜSSEN …
Nachdem wir Lists und Sets gesehen haben, verwenden wir jetzt eine Map
Endlich wieder zurück zu Generics
Lösung zur Übung
12 Lambdas und Streams: Was – nicht wie!
Sagen Sie dem Computer, WAS Sie wollen
Wenn for-Schleifen schieflaufen
Einführung in die Streams-API
Ergebnisse von einem Stream erhalten
Richtlinien für die Arbeit mit Streams
Hallo Lambda, mein (nicht ganz so) alter Freund
Erkennen Sie funktionale Interfaces
Lous Herausforderung Nr. 1: Finde alle »Rock«-Songs
Lous Herausforderung Nr. 2: Alle Genres auflisten
Übungen
13 Riskantes Verhalten
Bauen wir eine Musikmaschine
Zuerst brauchen wir einen Sequencer
Eine Exception ist ein Objekt … des Typs Exception
Flusskontrolle in try/catch-Blöcken
Eine Methode kann mehr als eine Exception werfen
Mehrfache catch-Blöcke müssen von klein nach groß geordnet werden
Ausweichen (durch Deklaration) verzögert nur das Unausweichliche
Codeküche
Version 1: Ihre allererste Soundplayer-App
Version 2: Kommandozeilenargumente für das Experimentieren mit Sounds
Übungen

14 Innen hui, außen GUI

Alles beginnt mit einem Fenster

Wie man an ein Benutzer-Event kommt

Listener, Quellen und Events

Machen Sie sich Ihr eigenes Grafik-Widget

Spaß mit paintComponent()

GUI-Layouts: mehrere Widgets in einen Frame packen

Die Rettung: Innere Klassen!

Rettung durch Lambdas!

Innere Klassen für Animationen einsetzen

Eine einfachere Möglichkeit, Messages und Events zu erstellen

Übungen

15 Mehr Schwung mit Swing

Swing-Komponenten

Layoutmanager

Die drei wichtigsten Layoutmanager: Border, Flow und Box

Es gibt keine Dummen Fragen

Spielen mit Swing-Komponenten

Codeküche

Die BeatBox

Übungen

16 Objekte (und Text) speichern

Ein serialisiertes Objekt in eine Datei schreiben

Soll eine Klasse serialisierbar sein, implementieren Sie Serializable

Deserialisierung: ein Objekt wiederherstellen

Version-ID: Ein großes Serialisierungsproblem

Einen String in ein Textdatei schreiben

Aus einer Textdatei lesen

Quiz Card Player (Code grob skizziert)
Path, Paths und Files (mit Verzeichnissen spielen)
Endlich, ein genauer Blick auf finally
Ein BeatBox-Pattern speichern
Übungen
17 Eine Verbindung herstellen
Verbinden, senden, empfangen
Der DailyAdviceClient (»Tipp des Tages«)
Ein einfaches Serverprogramm schreiben
Java hat mehrere Threads, aber nur eine Thread-Klasse
Die drei Zustände eines neuen Threads
Einen Thread schlafen schicken
Zwei (oder mehr!) Threads erzeugen und starten
Feierabend am Thread-Pool
Der neue und verbesserte SimpleChatClient
Übungen
18 Nebenläufigkeitsprobleme behandeln
Das Ryan-und-Monica-Problem, in Code ausgedrückt
Das Schloss eines Objekts verwenden
Das gefürchtete »Problem der verlorenen Aktualisierung«
Die increment()-Methode atomar machen. Synchronisieren Sie sie!
Blockade, eine tödliche Seite der Synchronisierung
»Vergleichen und tauschen« mit atomaren Variablen
Immutable Objekte verwenden
Mehr Probleme mit geteilten Daten
Verwenden Sie eine threadsichere Datenstruktur
Übungen
A Anhang A

Das endgültige BeatBox-Clientprogramm
Das endgültige BeatBox-Serverprogramm
B Anhang B
#11 JShell (Java REPL)
#10 Packages
#9 Immutabilität in Strings und Wrappern
#8 Zugriffsebenen und Zugriffsmodifier (wer sieht was)
#7 Varargs
#6 Annotationen
#5 Lambdas und Maps
#4 Parallele Streams
#3 Enumerations (enumerierte Typen/Enums)
#2 Lokale Typinferenz für Variablen (var)
#1 Records
i Index

4 Methoden benutzen Instanzvariablen

Wie sich Objekte verhalten

Zustand beeinflusst Verhalten, Verhalten beeinflusst Zustände. Wir wissen, dass Objekte über **Zustand** und **Verhalten** verfügen, die durch **Instanzvariablen** und **Methoden** repräsentiert werden. Dabei haben wir uns noch nicht angesehen, welche *Beziehung* Zustand und Verhalten zueinander haben. Wir wissen bereits, dass jede Instanz einer Klasse (jedes Objekt eines bestimmten Typs) seine eigenen, einmaligen Werte für Instanzvariablen besitzen kann. Hund (Dog-Objekt) A kann einen *Namen* haben, »Fido«, und ein *Gewicht* von 35 Kilo. Hund B heißt »Killer« und wiegt 5 Kilo. Wenn nun die Dog-Klasse eine Methode namens makeNoise() (gibLaut()) besitzt, sollte das Bellen eines 35-kg-Hunds dann nicht etwas tiefer sein als das seines deutlich kleineren Artgenossen? (Wobei wir davon ausgehen, dass ein nerviges Kläffen tatsächlich als *Bellen* durchgeht.) Glücklicherweise ist genau das der springende Punkt eines Objekts – es besitzt *Verhalten*, das sich auf seinen *Zustand* auswirkt. Anders gesagt, ***Methoden benutzen die Werte von Instanzvariablen***. Ein Beispiel: »Wenn der Hund weniger als 7 Kilo

wiegt, erzeuge ein Kläffgeräusch, sonst …« oder »erhöhe das Gewicht um 5«. ***Also los, lassen Sie uns ein paar Zustände ändern.***

Erinnern Sie sich: Eine Klasse beschreibt, was ein Objekt weiß und was es tut.

Eine Klasse ist der Bauplan für ein Objekt. Wenn Sie eine Klasse schreiben, sagen Sie der JVM, wie sie ein Objekt dieses Typs herstellen soll. Sie wissen bereits, dass solch ein Objekt verschiedene Werte für *Instanzvariablen* haben kann. Was ist aber mit den Methoden?

Kann jedes Objekt dieses Typs unterschiedliches Methodenverhalten haben?

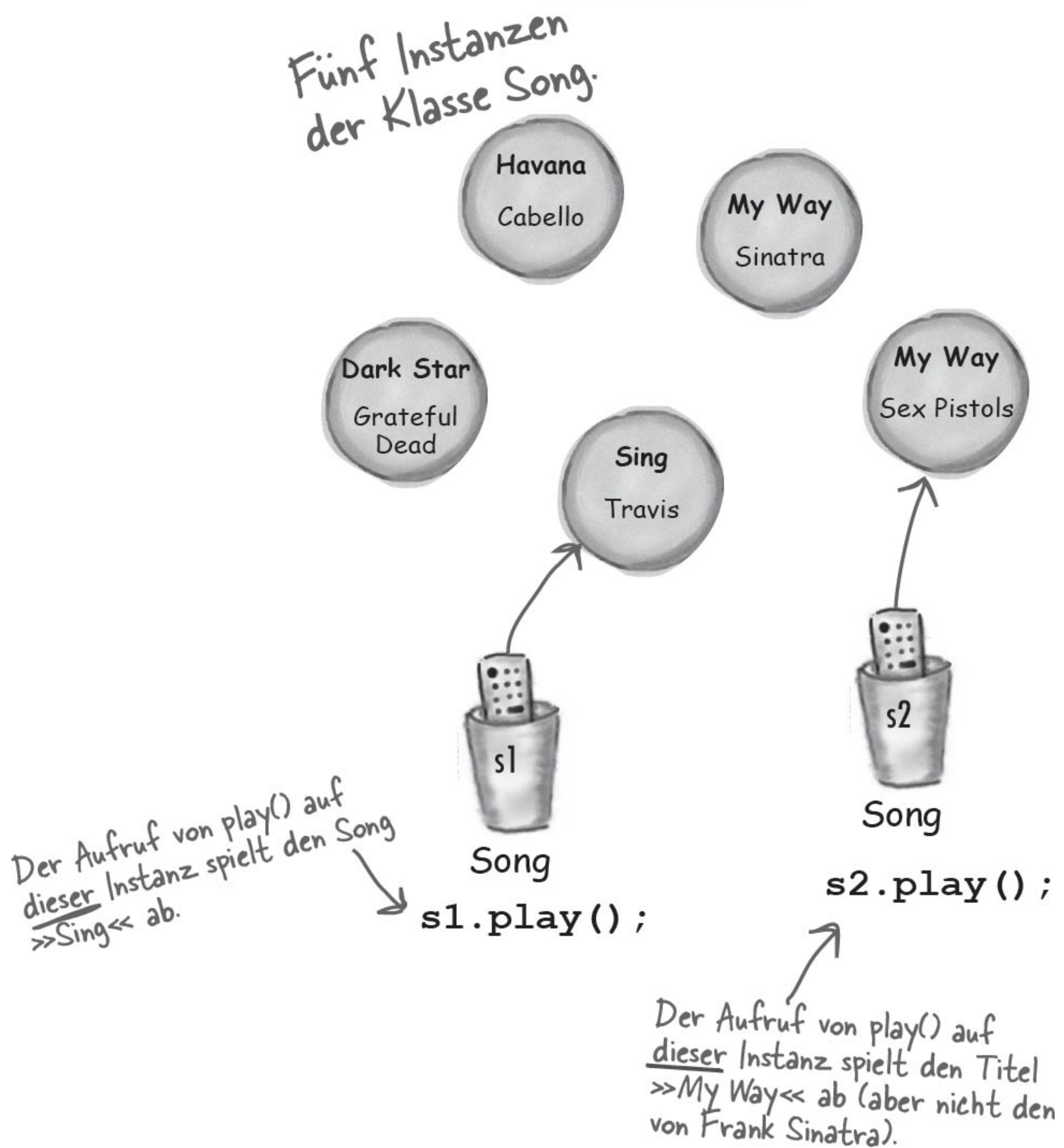
Na ja … ***irgendwie schon.****

Jede Instanz einer bestimmten Klasse besitzt die gleichen Methoden, aber die Methoden können sich abhängig von den Werten der Instanzvariablen unterschiedlich *verhalten*.

Die Klasse Song besitzt die beiden Instanzvariablen *title* (Titel) und *artist* (Künstler). Wenn Sie auf einer Instanz die play()-Methode aufrufen, wird sie den Song abspielen, der durch den Wert der Instanzvariablen *title* und *artist* dieser Instanz dargestellt wird. Rufen Sie also die play()-Methode auf einer Instanz auf, hören Sie das Stück »Havana« von Cabello, während eine andere Instanz »Sing« von Travis spielt. Dabei bleibt der Code der Methode gleich.

```
  void play() {

      soundPlayer.playSound(title, artist);

  }


Song song1 = new Song();

song1.setArtist("Travis");

song1.setTitle("Sing");
```

```
Song song2 = new Song();

song2.setArtist("Sex Pistols");

song2.setTitle("My Way");
```



Die Größe beeinflusst das Bellen

Das Bellen eines kleinen Hunds ist anders als das eines großen Hunds.

Die *bark()*-Methode benutzt die Instanzvariable *size* der Klasse Dog, um zu entscheiden, wie der Hund bellen soll.

Dog
size

name
bark()

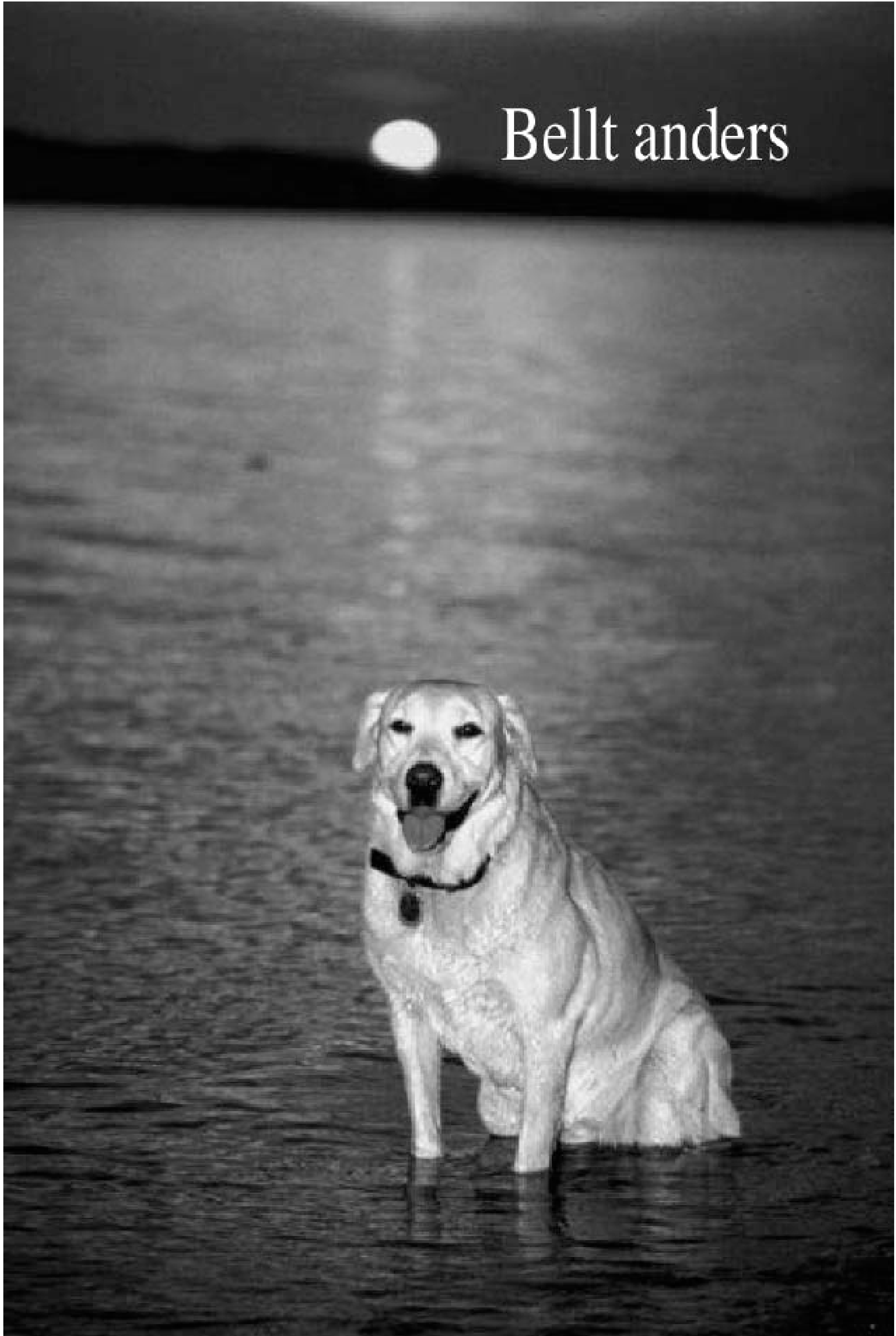
```
class Dog {

  int size;

  String name;

  void bark() {

    if (size > 60) {

      System.out.println("Wooof! Wooof!");

    } else if (size > 14) {

      System.out.println("Ruff! Ruff!");

    } else {

      System.out.println("Yip! Yip!");

    }

  }

}
```

```
class DogTestDrive {
```

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

    Dog one = new Dog();

    one.size = 70;

    Dog two = new Dog();

    two.size = 8;

    Dog three = new Dog();

    three.size = 35;

    one.bark();

    two.bark();

    three.bark();

  }

}
```

Datei Bearbeiten Fenster Hilfe TotStellen

```
%java DogTestDrive
Wooof! Wooof!
Yip! Yip!
Ruff!  Ruff!
```


Bellt anders

Sie können einer Methode Informationen schicken

Wie von einer Programmiersprache zu erwarten, können Sie Ihren Methoden Werte übergeben. Um einem Dog-Objekt zu sagen, wie oft es bellen soll, könnten Sie beispielsweise schreiben:

```
d.bark(3);
```

Abhängig von Ihrem Programmierhintergrund und Ihren persönlichen Vorlieben benutzen *Sie* vermutlich einen Begriff wie *Argumente* oder *Parameter* für die an eine Methode übergebenen Werte. Zwar gibt es in der Informatik formale Unterschiede, die für Menschen in Laborkitteln wichtig sind (und die wohl kaum dieses Buch lesen), wir haben in diesem Buch aber größere Fische zu fangen. *Sie* können sie nennen, wie Sie wollen (Argumente, Donuts, Fussel oder auch … Klaus?), aber wir machen es wie folgt:

Ein Aufrufer übergibt ein oder mehrere Argumente.

Eine Methode übernimmt einen oder mehrere Parameter.

Argumente sind die Dinge, die Sie an Methoden übergeben. Ein ***Argument*** (ein Wert wie 2, Foo oder eine Referenz auf ein Dog-Objekt) landet kopfüber in einem – genau – ***Parameter***. Und ein Parameter ist nichts anderes als eine lokale Variable. Eine Variable, deren Typ und Name im Körper der Methode verwendet werden kann.

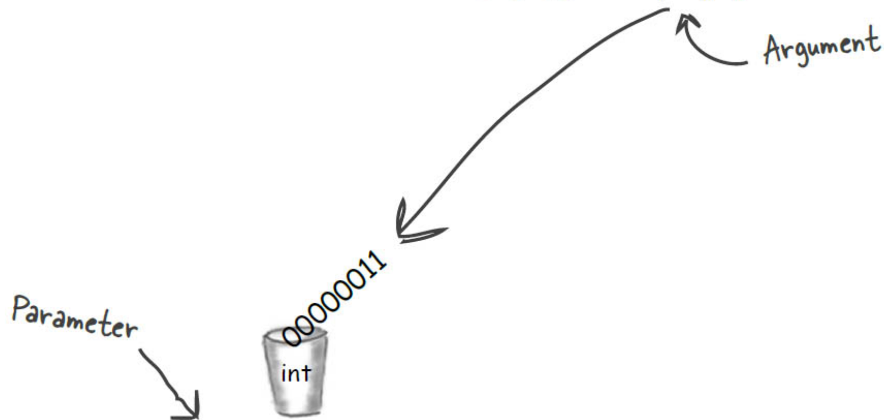
Aber hier kommt der wichtige Teil: **Übernimmt eine Methode einen Parameter, *müssen* Sie ihr beim Aufruf etwas übergeben.** Und dieses Etwas muss ein Wert des richtigen Typs sein.

1. Ruft die bark-Methode auf der Dog-Referenz auf und übergibt ihr den Wert 3 (als Argument der Methode).

2. Die Bits, die den int-Wert 3 repräsentieren, werden an die bark-Methode übergeben.

3. Die Bits landen im numOfBarks-Parameter (einer Variablen mit int-Größe).

❹ Verwendet den Parameter numOfBarks als Variable im Code der Methode.




```
void bark(int numOfBarks) {
  while (numOfBarks > 0) {
    System.out.println("ruff");
    numOfBarks = numOfBarks - 1;
  }
}
```

Sie können von Methoden etwas *zurückbekommen*

Methoden können auch *Rückgabewerte* haben. Jede Methode wird mit einem Rückgabetyp deklariert, bisher hatten unsere Methoden jedoch alle den Rückgabetyp **void**, was bedeutet, dass sie nichts zurückgeben.

```
void go() {

{
```

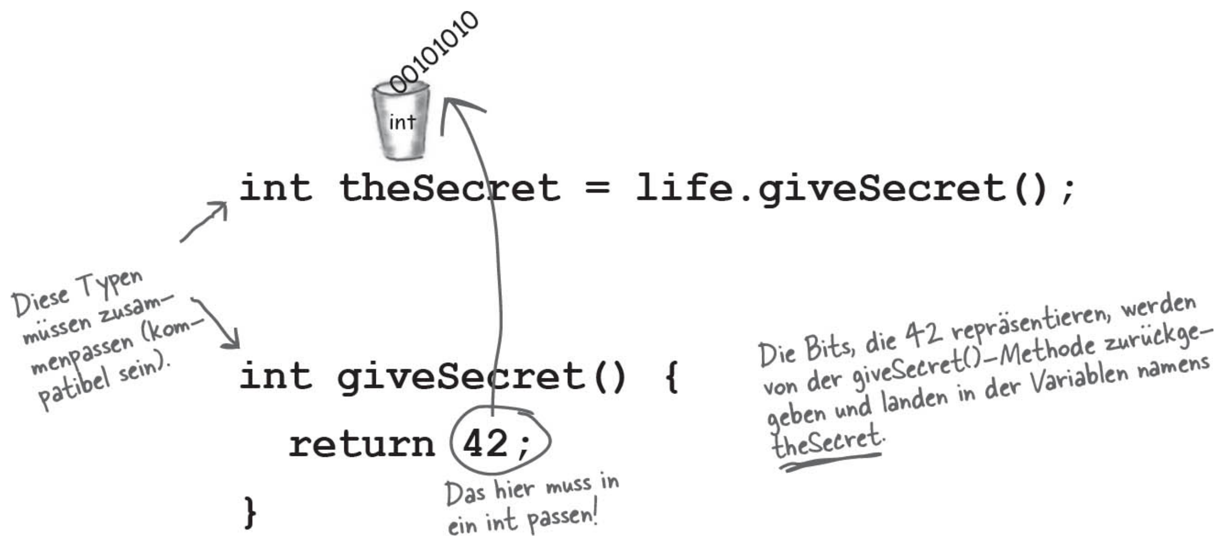
Aber wir können eine Methode so deklarieren, dass sie dem Aufrufer einen Wert zurückgibt, wie hier:

```
int giveSecret() {

  return 42;

}
```

Wenn Sie eine Methode so deklarieren, dass sie einen Wert zurückgibt, muss dieser Wert den deklarierten Typ haben! (Oder einen Wert, der mit dem deklarierten Typ *kompatibel* ist. Wir werden hierauf näher eingehen, wenn wir in den Kapiteln 7 und 8 auf Polymorphie zu sprechen kommen.)

Was immer Sie zurückzugeben versprechen, sollten Sie auch zurückgeben!

Der Compiler verhindert, dass Sie Daten des falschen Typs zurückgeben.



Sie können einer Methode mehr als eine Sache übergeben

Methoden können mehrere Parameter haben. Diese werden bei der Deklaration durch Kommata voneinander getrennt. Ebenso werden die Argumente bei der Übergabe durch Kommata getrennt. Am wichtigsten ist aber: Wenn eine Methode Parameter besitzt, *müssen* die Argumente den richtigen Typ und die richtige Reihenfolge haben.

Eine Zwei-Parameter-Methode aufrufen und ihr zwei Argumente übergeben.

```
void go() {
  TestStuff t = new TestStuff();
  t.takeTwo(12, 34);
}
```

Die übergebenen Argumente landen in der gleichen Reihenfolge in der Methode, in der sie übergeben wurden. Das erste Argument landet im ersten Parameter, das zweite Argument im zweiten Parameter und so weiter.

```
void takeTwo(int x, int y) {
  int z = x + y;
  System.out.println("Total is " + z);
}
```

Sie können einer Methode auch eine Variable übergeben, solange der Variablentyp mit dem entsprechenden Parametertyp übereinstimmt.

```
void go() {
   int foo = 7;
   int bar = 3;
   t.takeTwo(foo, bar);
}
```

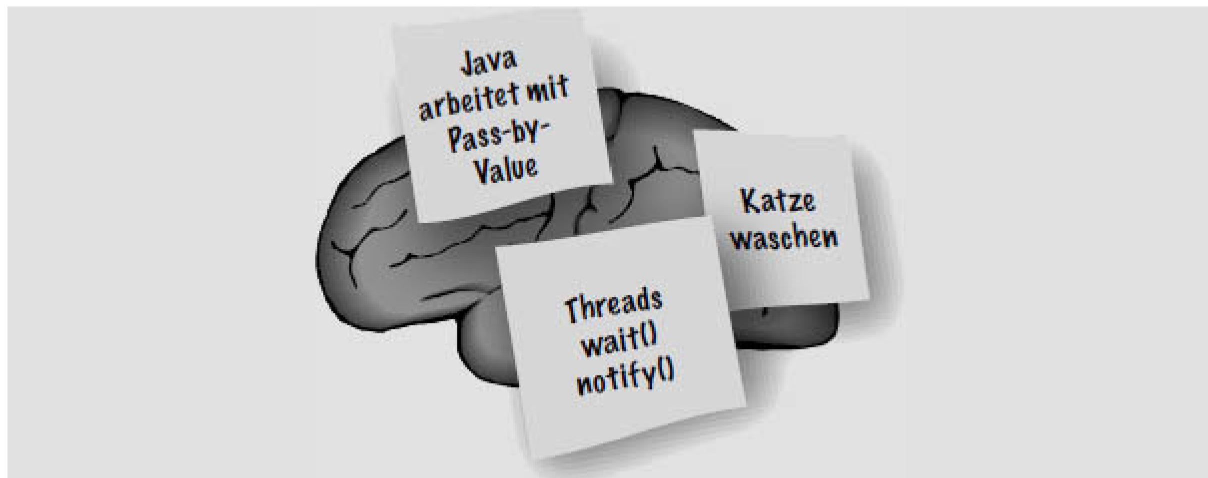
Die Werte von foo und bar landen in den Parametern x und y. Die Bits in x sind jetzt also identisch mit denen in foo (dem Bitmuster für die Ganzzahl 7), und die Bits in y sind identisch mit den Bits in bar.

```
void takeTwo(int x, int y) {
   int z = x + y;
   System.out.println("Total is " + z);
}
```

Was ist der Wert von z? Das Ergebnis entspricht dem, das Sie erhalten, wenn Sie foo + bar bei der Übergabe an die takeTwo-Methode addieren.

Java ist Pass-by-Value.

Das bedeutet Pass-by-Copy.



Damit es hängen bleibt.

Rosen sind rot,

dieser Vers ist für Sie,

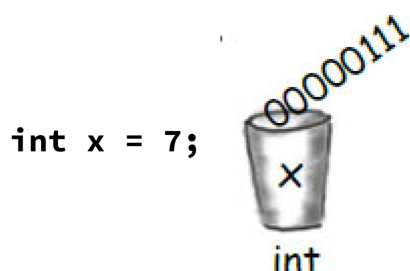
übergeb ich 'nen Wert

ist das 'ne Kopie.

Sie meinen, Sie könnten das besser? Dann mal los! Ersetzen Sie die blöde zweite Zeile mit Ihrer eigenen. Oder noch besser, ersetzen Sie das ganze Gedicht durch Ihr eigenes, damit Sie es nie wieder vergessen.

1. Deklariert eine int-Variable und weist ihr den Wert »7« zu. Das Bitmuster für 7 landet in der Variablen mit dem Namen x.

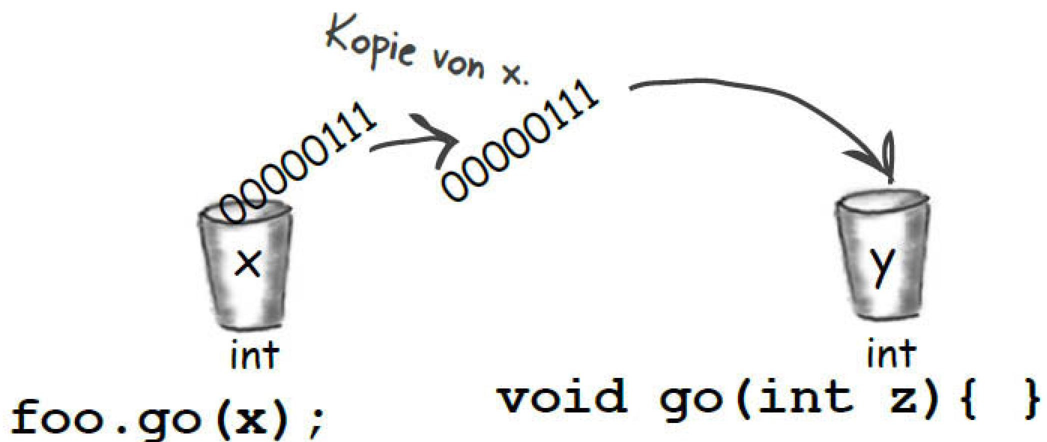
```
int x = 7;
```



2. Deklariert eine Methode mit einem int-Parameter namens z.

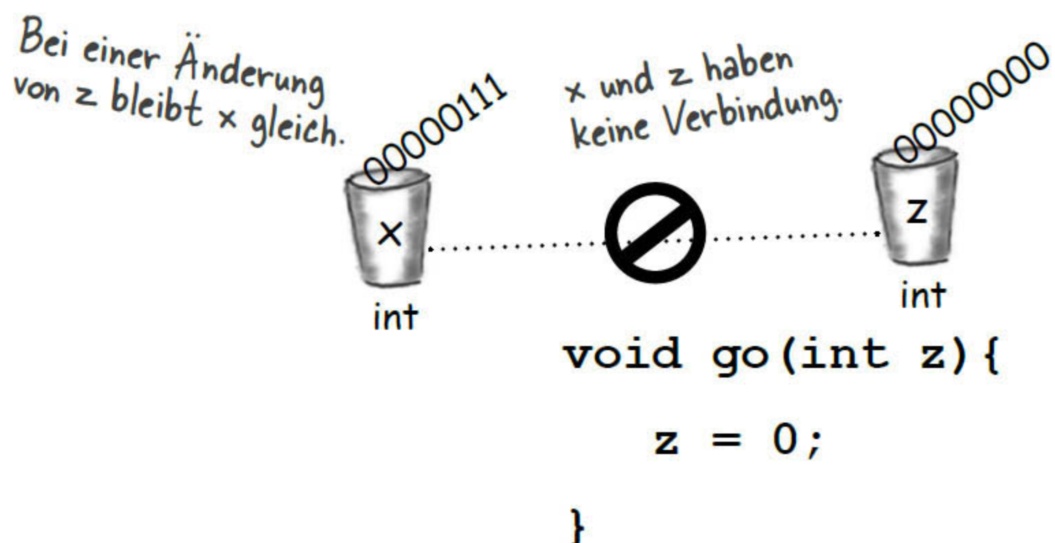
```
void go(int z){ }
```

3 Ruft die go()-Methode auf und übergibt die Variable x als Argument. Die Bits in x werden kopiert, und die Kopie landet in z.



4 Ändert innerhalb der Methode den Wert von z. Der Wert von x bleibt dabei gleich! Das an den Parameter z übergebene Argument war nur eine Kopie von x.

Die Methode kann die Bits in der aufrufenden Variablen x nicht ändern.



Es gibt keine

Dummen Fragen

F: **Was passiert, wenn man statt eines elementaren Typs ein Objekt als Argument übergeben will?**

A: Dazu werden Sie in späteren Kapiteln mehr erfahren. Eigentlich *wissen* Sie die Antwort jedoch schon. Java übergibt *alles* by Value, also als Wert. ***Alles***. Aber … Wert heißt *Bits in einer Variablen*. Und wie Sie wissen, kann man keine Objekte in Variablen packen. Die Variable ist eine Fernbedienung – *eine Referenz auf ein Objekt*. Wenn Sie eine Referenz auf ein Objekt an eine Methode übergeben, übergeben Sie also eine *Kopie der Fernbedienung*. Bleiben Sie dran – dazu haben wir noch viel mehr zu sagen.

F: **Kann eine Methode mehrere Rückgabewerte deklarieren? Oder besteht eine Möglichkeit, mehr als einen Wert zurückgeben?**

A: Mehr oder minder. Eine Methode kann nur einen Rückgabewert deklarieren. ABER … wenn Sie, sagen wir, drei int-Werte zurückgeben wollen, kann der deklarierte Rückgabewert ein int-*Array* sein. Packen Sie die ints in das Array und geben Sie es zurück. Wenn Sie mehrere Werte mit unterschiedlichen Typen zurückgeben möchten, ist das etwas komplizierter. Dazu kommen wir später, wenn wir über ArrayList reden.

F: **Muss ich genau den Typ zurückgeben, den ich deklariert habe?**

A: Sie können alles zurückgeben, das *implizit* in den deklarierten Typ umgewandelt werden kann. Sie könnten also einen byte-Wert zurückgeben, obwohl ein int-Wert erwartet wird. Dem Aufrufer ist das egal, weil der byte-Wert problemlos in die int-Variable passt, die der Aufrufer verwendet, um das Ergebnis zu speichern. Ist der deklarierte Typ dagegen *kleiner* als der Typ des gewünschten Rückgabewerts, müssen Sie eine *explizite* Umwandlung vornehmen (wie das funktioniert, sehen wir in Kapitel 5).

F: **Muss ich mit dem Rückgabewert einer Methode etwas machen? Kann ich ihn einfach ignorieren?**

A: In Java müssen Sie einen Rückgabewert nicht zwingend weiterbenutzen. Manchmal möchten Sie vielleicht eine Methode aufrufen, deren Rückgabetyp nicht void ist, obwohl der Rückgabewert Sie gar nicht interessiert. In einem solchen Fall rufen Sie die Methode wegen der Arbeit auf, die *innerhalb* der Methode getan wird, nicht wegen des Ergebnisses, das die Methode *zurückgibt*. In Java müssen Sie den Rückgabewert weder zuweisen noch weiterverwenden.

Zur Erinnerung: Java sind die Typen wichtig!

Sie können keine Giraffe zurückgeben, wenn der Rückgabetyp als Kaninchen deklariert ist. Das gilt auch für Parameter. Sie können keine Giraffe übergeben, wenn die Methode ein Kaninchen übernimmt.

PUNKT FÜR PUNKT

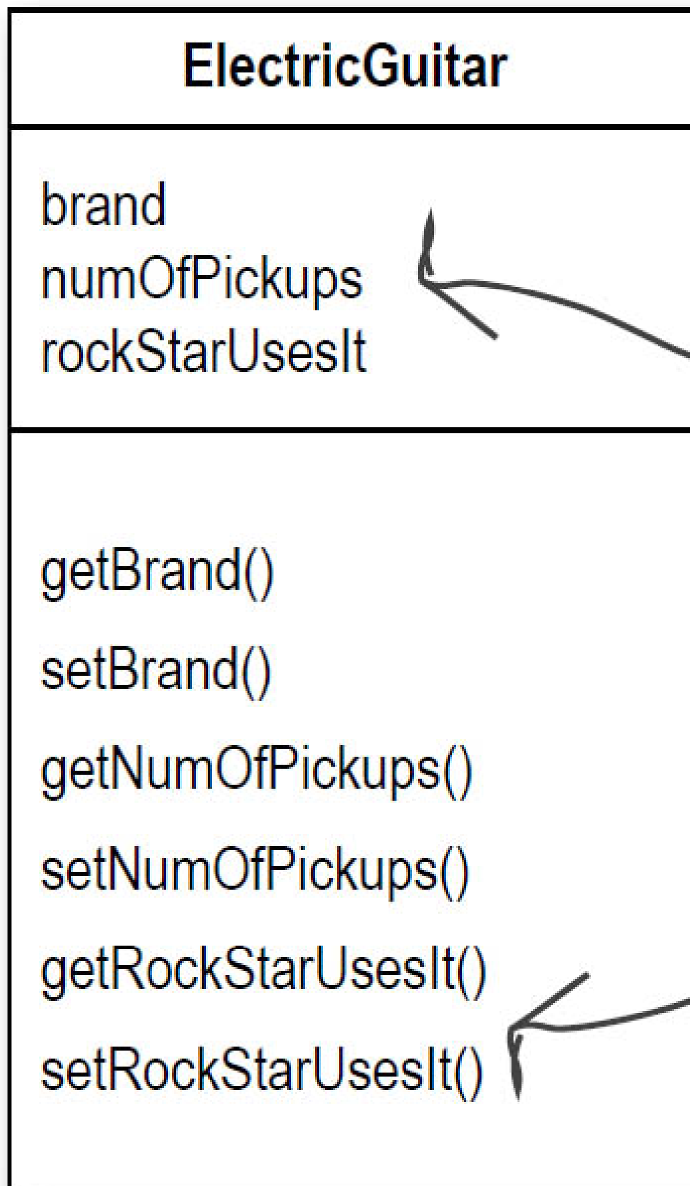
- Klassen definieren, was ein Objekt weiß und was es tut.
- Dinge, die ein Objekt weiß, heißen **Instanzvariablen** (Zustand).
- Dinge, die ein Objekt tut, sind seine **Methoden** (Verhalten).
- Methoden verwenden Instanzvariablen, damit sich Objekte des gleichen Typs unterschiedlich verhalten können.
- Eine Methode kann Parameter haben. Das heißt, Sie können der Methode einen oder mehrere Werte übergeben.
- Anzahl und Typen der übergebenen Werte müssen mit der Reihenfolge und den Typen der von der Methode deklarierten Parameter übereinstimmen.

- An Methoden übergebene und von Methoden zurückgegebene Werte können implizit zu einem größeren Typ oder explizit in einen kleineren Typ umgewandelt werden. Im zweiten Fall spricht man von *Type Casting* (Typumwandlung).
- Der als Argument an eine Methode übergebene Wert kann ein literaler Wert (wie 2, 'c' etc.) oder eine Variable des deklarierten Parametertyps sein (zum Beispiel *x,* wobei *x* eine int-Variable ist). (Es gibt noch mehr Dinge, die Sie als Argumente übergeben können, aber darum kümmern wir uns später.)
- Eine Methode *muss* einen Rückgabetyp deklarieren. Der Rückgabetyp void bedeutet, dass die Methode nichts zurückgibt.
- Deklariert eine Methode einen Rückgabetyp, der nicht void ist, *muss* der Rückgabetyp mit dem deklarierten Rückgabetyp kompatibel sein.

Coole Dinge, die Sie mit Parametern und Rückgabetypen anstellen können

Nachdem Sie nun wissen, wie Parameter und Rückgabetypen funktionieren, ist es an der Zeit, sie sinnvoll zu nutzen: als **Getter** und **Setter**. Wenn Sie das ganz formal ausdrücken wollen, können Sie auch *Akzessoren* und *Mutatoren* dazu sagen. Aber das hieße, Silben zu verschwenden. Außerdem entsprechen Getter und Setter den Java-Namenskonventionen. Also bleiben wir auch dabei.

Mit Gettern und Settern können Sie Dinge *auslesen* (engl. to get) und *schreiben* (engl. to set), normalerweise die Werte von Instanzvariablen. Der einzige Lebenszweck eines Getters ist es, den Wert dessen, was dieser bestimmte Getter da auslesen soll, als Rückgabewert zurückzuliefern. Und daher überrascht es Sie vermutlich kaum, dass ein Setter dafür lebt und atmet, einen Argumentwert zu übernehmen und ihn zu benutzen, um den Wert einer Instanzvariablen zu *setzen* (festzulegen).




```
class ElectricGuitar {

  String brand;

  int numOfPickups;

  boolean rockStarUsesIt;

  String getBrand() {
```

```
    return brand;
  }
  void setBrand(String aBrand) {
    brand = aBrand;
  }
  int getNumOfPickups() {
    return numOfPickups;
  }
  void setNumOfPickups(int num) {
    numOfPickups = num;
  }
  boolean getRockStarUsesIt() {
    return rockStarUsesIt;
  }
  void setRockStarUsesIt(boolean yesOrNo) {
    rockStarUsesIt = yesOrNo;
  }
```

```
}
```

Kapselung

Tun Sie es oder machen Sie sich auf Hohn und Spott gefasst.

Bis zu diesem wirklich wichtigen Moment haben wir eine der schlimmsten OO-Sünden begangen (und wir reden hier nicht von einem kleinen Missgeschick, wie ohne Getränk auf einer Bottleparty aufzutauchen). Wir reden hier von einer der Sieben Todsünden.

Unsere schändliche Missetat?

Unsere Daten zu entblößen!

Hier sitzen wir, summen sorglos ein Lied vor uns hin und lassen unsere Daten einfach so herumliegen, wo *jeder* sie sehen und sogar anfassen kann.

Vielleicht kennen Sie dieses leicht unangenehme Gefühl schon, das einen beschleicht, wenn man seine Instanzvariablen entblößt. Und damit meinen wir, dass sie über den Punktoperator erreichbar sind, wie in:

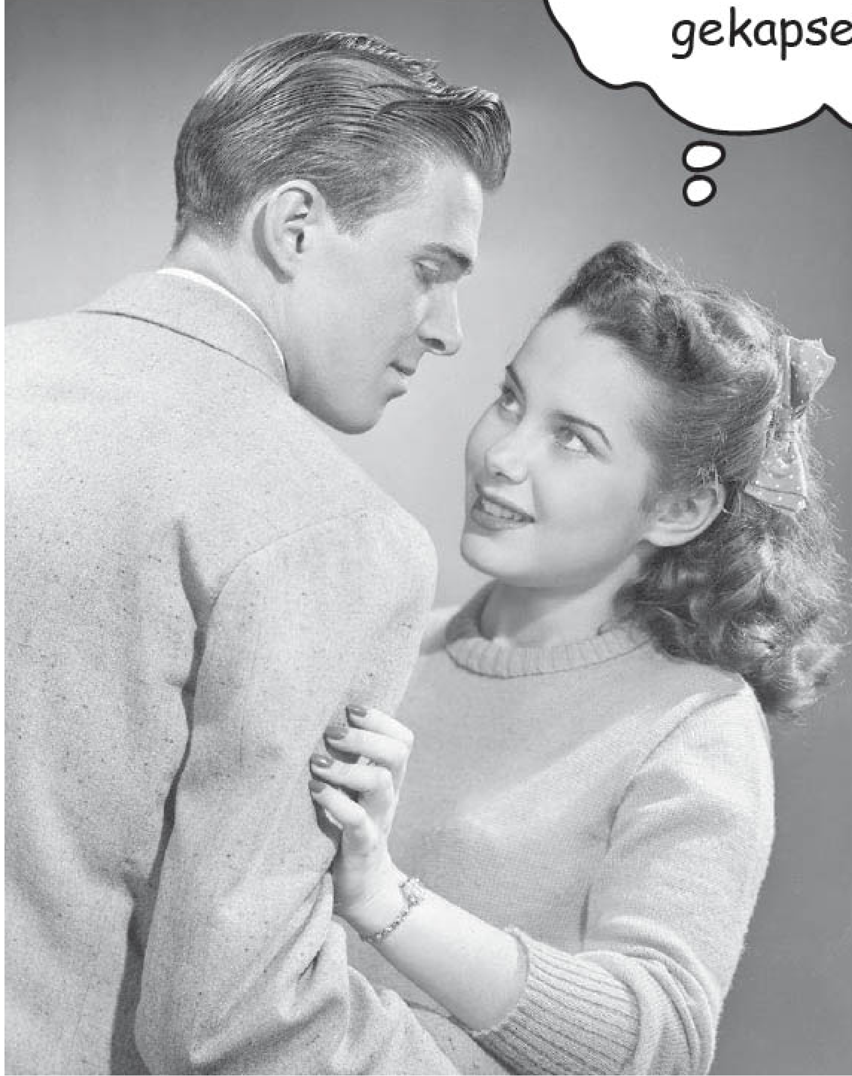
```
theCat.height = 27;
```

Stellen Sie sich vor, Sie könnten Ihre Fernbedienung benutzen, um eine direkte Änderung an der Instanzvariablen size eines Cat-Objekts vorzunehmen. In den Händen der falschen Person, ist eine Referenzvariable (Fernbedienung) eine gefährliche Waffe. Denn was gilt es zu verhindern:

```
theCat.height = 0;
```

Du meine Güte! Das darf auf keinen Fall passieren!

Das wäre eine wirklich üble Geschichte. Am besten, Sie schreiben Setter-Methoden für alle Instanzvariablen und finden eine Möglichkeit, anderen Code zu zwingen, die Setter-Methoden aufzurufen, anstatt direkt auf die Daten zuzugreifen.


Jenny sagt, dass
du ordentlich
gekapselt bist ...

Indem wir erzwingen, dass eine Setter-Methode aufgerufen wird, können wir die Katze (Cat-Objekt) vor inakzeptablen Größenänderungen bewahren.

```
public void setHeight(int ht) {
    if (ht > 9) {
        height = ht;
    }
}
```

Wir bauen Tests ein, die eine minimale Katzengröße sicherstellen.

Daten verstecken

Ja, *so* einfach kommt man von einer Implementierung, die nur auf falsche Daten wartet, zu einer Version, die Ihre Daten schützt und Ihnen *trotzdem* die Möglichkeit offenhält, die Implementierung später noch zu ändern.

Aber wie genau versteckt man Daten? Mit den Zugriffsmodifiern **`public`** und **`private`**. Den Modifier **`public`** kennen Sie bereits, wir haben ihn bei allen main()-Methoden verwendet.

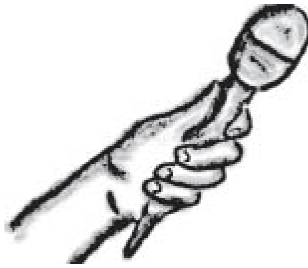
Hier eine Faustregel für *Kapselungsanfänger* (natürlich gelten alle Garantiebeschränkungen für Faustregeln): Markieren Sie Ihre Instanzvariablen als ***private*** und stellen Sie öffentliche (***public***) Getter und Setter für die Zugriffskontrolle bereit. Sobald Sie sich mit Entwürfen und der Programmierung in Java besser auskennen, werden Sie die Sache vermutlich etwas anders angehen. Im Moment ist dieser Weg für Sie aber der sicherste.

Markieren Sie Instanzvariablen als private.

Markieren Sie Getter und Setter als public.

»Leider hat Bill vergessen, seine Cat-Klasse zu kapseln. Jetzt müssen wir uns mit einer platten Katze herumschlagen.«

(bei der Kaffeemaschine aufgeschnappt)



Java im Gespräch

Interview der Woche: Die Wahrheit über Kapselung. Ein Objekt klärt auf.

Von Kopf bis Fuß: Was ist an der Kapselung denn so besonders?

Objekt: Kennen Sie den Traum, in dem Sie vor 500 Menschen einen Vortrag halten und plötzlich feststellen, dass Sie *nackt* sind?

Von Kopf bis Fuß: Oh ja. Der ist so ähnlich wie der Traum mit dem Pilates-Gerät. Müssen wir nicht noch mal haben. Sie fühlen sich also nackt. Aber was ist daran so gefährlich, außer dass man ein bisschen von sich preisgibt?

Objekt: Gefährlich? Ist das *gefährlich*? [lacht] Hey Instanzen, habt ihr das gehört? *»Was ist daran so gefährlich?«*, will er wissen. [kippt vor Lachen vom Stuhl]

Von Kopf bis Fuß: Was ist denn so lustig? Das scheint mir doch eine vernünftige Frage zu sein.

Objekt: Also gut. Ich erkläre es Ihnen. Es ist … [fängt wieder an, unkontrolliert zu lachen]

Von Kopf bis Fuß: Soll ich Ihnen ein Glas Wasser holen?

Objekt: Puh, Junge. Nein, es geht schon wieder. Ich bin jetzt ernst. Tief atmen. Also, fangen Sie an.

Von Kopf bis Fuß: Und, wovor schützt Sie die Kapselung?

Objekt: Die Kapselung umgibt meine Instanzvariablen mit einem Kraftfeld, sodass niemand sie auf – sagen wir mal – etwas *Unangemessenes* setzen kann.

Von Kopf bis Fuß: Können Sie mir ein Beispiel geben?

Objekt: Aber gern. Bei der Programmierung der meisten Instanzvariablen macht man Annahmen bezüglich der Wertgrenzen. Denken Sie beispielsweise über all die Dinge nach, die kaputtgehen würden, wenn negative Zahlen erlaubt würden. Die Anzahl von Toiletten in einem Büro. Die Geschwindigkeit eines Flugzeugs. Hantelgewichte. Handynummern. Temperatureinstellung auf einer Mikrowelle.

Von Kopf bis Fuß: Ich verstehe. Aber wie kann die Kapselung Grenzen setzen?

Objekt: Indem sie anderen Code zwingt, Setter-Methoden zu benutzen. So kann die Setter-Methode die Parameter überprüfen und entscheiden, ob etwas machbar ist. Vielleicht weist die Methode den Parameter zurück und macht gar nichts, oder sie löst eine Exception aus (zum Beispiel wenn Sie eine Kreditkarte beantragen, aber kein Geld auf dem Konto haben). Vielleicht rundet die Methode den übergebenen Parameter aber auch auf den nächsten akzeptablen Wert. Der Punkt ist, dass Sie in der Setter-Methode tun können, was Sie wollen, während Sie *überhaupt nichts* tun können, wenn Ihre Instanzvariablen öffentlich sind.

Von Kopf bis Fuß: Aber manchmal sehe ich Setter-Methoden, die einfach den Wert schreiben (setzen), ohne irgendetwas zu überprüfen. Wenn Sie eine Instanzvariable ohne Begrenzung haben, schafft die Setter-Methode doch nur unnötigen Aufwand, oder? Vielleicht sorgt sie sogar für Leistungseinbußen.

Objekt: Bei Settern (und Gettern) geht es darum, dass Sie ***Ihre Meinung später noch ändern können, ohne den Code anderer kaputtzumachen!*** Stellen Sie sich vor, die halbe Belegschaft Ihres Unternehmens würde Ihre Klasse mit öffentlichen Instanzvariablen verwenden. Eines Tages stellen Sie fest: »Hoppla, da gibt es etwas, das ich bei dem Wert nicht eingeplant habe, ich muss da doch eine Setter-Methode einbauen …« Damit machen Sie den Code aller anderen kaputt. Das Coole an der Kapselung ist, dass *Sie Ihre Meinung ändern können*, ohne dass jemand verletzt wird. Der Leistungsgewinn bei einer direkten Nutzung von Variablen ist gering und ist es *so gut wie nie* wert, auf Setter zu verzichten.

Die GoodDog-Klasse kapseln

GoodDog
size
getSize( ) setSize( ) bark( )

```java
class GoodDog {
  private int size;

  public int getSize() {
    return size;
  }

  public void setSize(int s) {
    size = s;
  }

  void bark() {
    if (size > 60) {
      System.out.println("Wooof! Wooof!");
    } else if (size > 14) {
      System.out.println("Ruff!  Ruff!");
    } else {
      System.out.println("Yip! Yip!");
    }
  }
}

class GoodDogTestDrive {

  public static void main(String[] args) {
    GoodDog one = new GoodDog();
    one.setSize(70);
    GoodDog two = new GoodDog();
    two.setSize(8);
    System.out.println("Dog one: " + one.getSize());
    System.out.println("Dog two: " + two.getSize());
    one.bark();
    two.bark();
  }
}
```

Die Instanzvariable als private markieren.

Die Getter- und Setter-Methoden als public markieren.

Das Gute daran ist, dass Sie Ihre Meinung später noch ändern können, selbst wenn die Methoden keine neue Funktionalität hinzufügen. So können Sie später zurückkommen und eine Methode sicherer, schneller, besser machen.

Überall dort, wo ein bestimmter Wert benutzt werden kann, kann ein Methodenaufruf verwendet werden, der einen Wert dieses Typs zurückgibt.

Anstelle von:

int x = 3 + 24;

können Sie sagen:

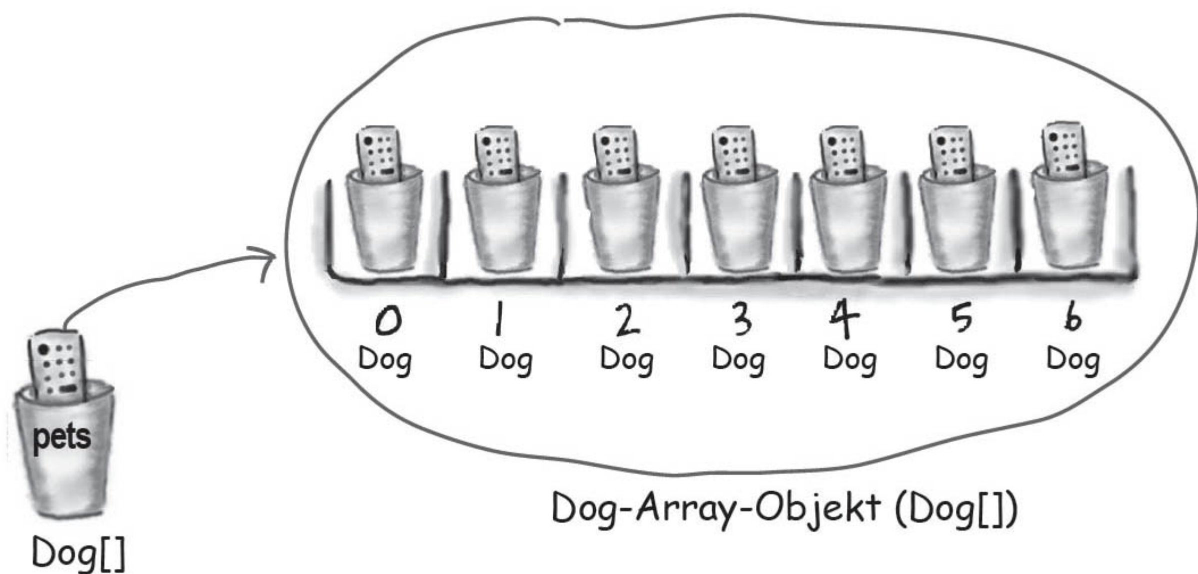
int x = 3 + one.getSize();

Wie verhalten sich Objekte in einem Array?

Wie jedes andere Objekt auch. Der einzige Unterschied besteht darin, wie Sie an die Objekte *herankommen*. Anders gesagt, wie Sie an die Fernbedienung kommen. Versuchen wir, ein paar Methoden auf Dog-Objekten in einem Array aufzurufen.

1. Ein Dog-Array deklarieren und erstellen, das sieben Dog-Referenzen enthalten soll.

```
Dog[] pets;

pets = new Dog[7];
```



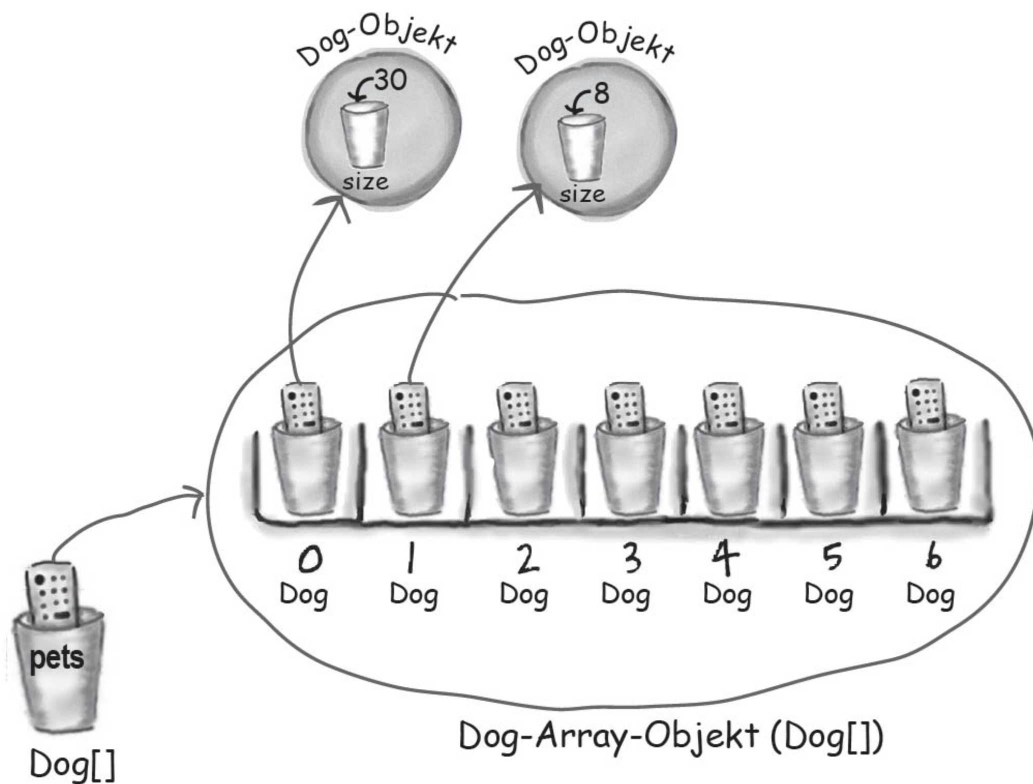
---

2. Zwei neue Dog-Objekte erstellen und sie den ersten beiden Array-Elementen zuweisen.

```
pets[0] = new Dog();

pets[1] = new Dog();
```

❸ Methoden auf den beiden Dog-Objekten aufrufen.

```
pets[0].setSize(30);

int x = pets[0].getSize();

pets[1].setSize(8);
```



Instanzvariablen deklarieren und initialisieren

Sie wissen bereits, dass für eine Variablendeklaration zumindest ein Name und ein Typ gebraucht werden:

```
int size;

String name;
```

Und Sie wissen auch, dass Sie die Variable gleichzeitig initialisieren (ihr einen Wert zuweisen) können.

```
int size = 420;

String name = "Donny";
```

Was passiert aber, wenn Sie eine Getter-Methode auf einer nicht initialisierten Variablen aufrufen? Anders gefragt: Was ist der *Wert* einer Instanzvariablen, *bevor* sie initialisiert wurde?

```
class PoorDog {
  private int size;
  private String name;

  public int getSize() {
    return size;
  }

  public String getName() {
    return name;
  }
}
```

Zwei Instanzvariablen deklarieren, aber keinen Wert zuweisen.

Was wird hier wohl zurückgegeben??

```
public class PoorDogTestDrive {
  public static void main(String[] args) {
    PoorDog one = new PoorDog();
    System.out.println("Dog size is " + one.getSize());
    System.out.println("Dog name is " + one.getName());
  }
}
```

Was meinen Sie? Wird das hier überhaupt kompiliert?

Da Instanzvariablen immer einen Standardwert haben, müssen sie nicht unbedingt initialisiert werden. Elementare Zahlen (inklusive char) erhalten 0, boolesche Variablen false und Objektreferenzen null.

(Erinnern Sie sich: null steht einfach nur für eine Fernbedienung, die nichts steuert oder auf irgendetwas programmiert ist. Eine Referenz, aber kein Objekt.)

Instanzvariablen erhalten grundsätzlich einen Standardwert. Selbst wenn Sie keinen expliziten Wert zuweisen oder eine Setter-Methode aufrufen, hat die Instanzvariable einen Wert.

Ganzzahlige Typen	**0**
Fließkommatypen	**0.0**
Boolesche Typen	**false**
Referenzen	**null**

Datei Bearbeiten Fenster Hilfe TierarztAnrufen

```
% java PoorDogTestDrive
Dog size is 0
Dog name is null
```

Der Unterschied zwischen Instanzvariablen und lokalen Variablen

1. **Instanz**variablen werden innerhalb einer Klasse, aber nicht innerhalb einer Methode deklariert.

```
class Horse {

  private double height = 15.2;

  private String breed;

  // mehr Code …

}
```

2. **Lokale** Variablen werden in einer Methode deklariert.

```
class AddThing {
  int a;
  int b = 12;

  public int add() {
    int total = a + b;
    return total;
  }
}
```

INSTANZvariablen

LOKALE Variablen

3 **Lokale** Variablen MÜSSEN vor der Verwendung initialisiert werden!

```
class Foo {
  public void go() {
    int x;
    int z = x + 3;
  }
}
```

Das wird nicht kompiliert. Sie können die Variable x zwar ohne Wert deklarieren, aber sobald Sie versuchen, sie zu VERWENDEN, flippt der Compiler aus.

Datei Bearbeiten Fenster Hilfe Huch!

```
% javac Foo.java
Foo.java:4: variable x might not have been initialized
      int z = x + 3;
1 error       ^
```

Lokale Variablen erhalten KEINEN Standardwert! Der Compiler beschwert sich, wenn Sie versuchen, eine lokale Variable zu verwenden, bevor sie initialisiert ist.

Es gibt keine

Dummen Fragen

F: **Was ist mit Methodenparametern? Gelten die Regeln für lokale Variablen auch für sie?**

A: Methodenparameter sind praktisch das Gleiche wie lokale Variablen – sie werden *innerhalb* der Methode deklariert (na ja, technisch gesehen in der *Argumentliste*) und nicht im *Körper* der Methode. Trotzdem sind es lokale Variablen und keine Instanzvariablen. Aber Methodenparameter sind niemals uninitialisiert. Deshalb erhalten Sie auch keinen Compilerfehler, der Ihnen sagt, dass eine Parametervariable vielleicht nicht initialisiert ist.

Stattdessen gibt Ihnen der Compiler eine Fehlermeldung, wenn Sie versuchen, eine Methode aufzurufen, ohne ihr die benötigten Argumente zu übergeben. Parameter sind also *immer* initialisiert, weil der Compiler garantiert, dass Methoden immer mit Argumenten aufgerufen werden, die den Parametern entsprechen. Die Argumente werden den Parametern (automatisch) zugewiesen.

Variablen vergleichen (elementare und Referenztypen)

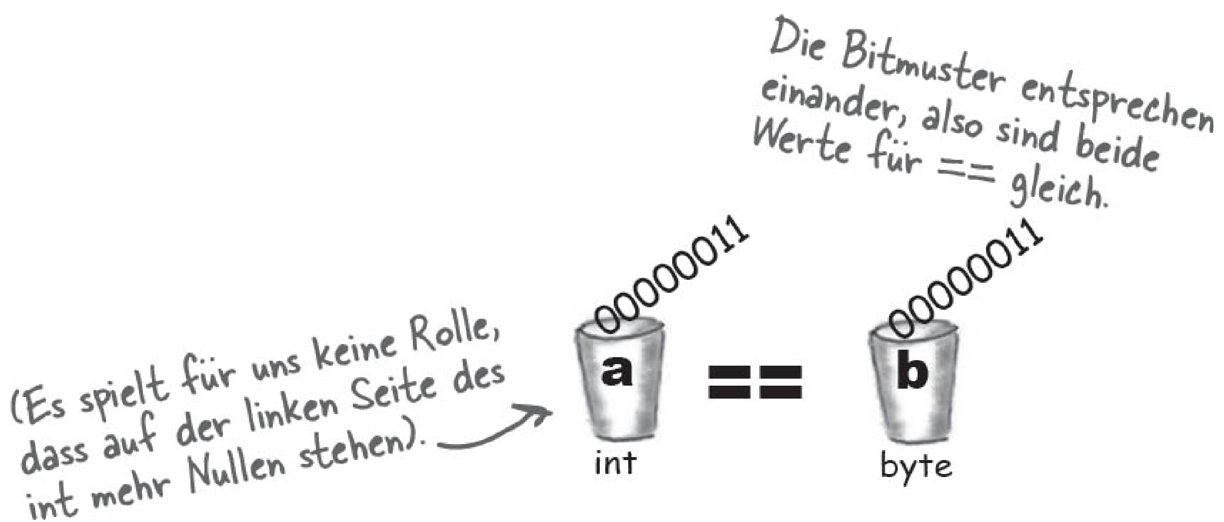
Manchmal wollen Sie wissen, ob zwei *elementare Werte* gleich sind. Vielleicht möchten Sie ein Ergebnis vom Typ int mit einem erwarteten Ganzzahlwert vergleichen. Das geht ganz einfach mit dem ==-Operator. In anderen Fällen wollen Sie herausfinden, ob zwei Referenzvariablen auf ein bestimmtes Objekt auf dem Heap verweisen, wie »Ist dieses Dog-Objekt das gleiche Dog-Objekt, mit dem ich begonnen habe?« Auch das ist leicht: Benutzen Sie einfach den ==-Operator. Gelegentlich wollen Sie aber wissen, ob zwei *Objekte* gleich sind. Dafür brauchen Sie die .equals()-Methode.

Was Gleichheit für Objekte tatsächlich bedeutet, hängt davon ab, um welchen Typ Objekt es jeweils geht. Enthalten zwei verschiedene String-Objekte etwa die gleichen Zeichen (zum Beispiel »mein Name«), sind sie gleichbedeutend, und zwar unabhängig davon, ob es zwei eigenständige Objekte auf dem Heap sind. Wie sieht es aber mit einem Dog-Objekt aus? Sollen zwei Dogs gleich behandelt werden, wenn sie die gleiche Größe und das gleiche Gewicht haben? Wahrscheinlich nicht. Ob zwei Objekte als gleich behandelt werden, hängt also davon ab, was für diesen bestimmten Objekttyp Sinn ergibt. Das Konzept der Objektgleichheit werden wir in späteren Kapiteln genauer betrachten. Im Moment müssen Sie nur verstehen, dass der ==-Operator *nur* verwendet wird, um die Bits in zwei Variablen zu vergleichen. *Was* diese Bits repräsentieren, spielt keine Rolle. Die Bits sind entweder gleich oder eben nicht.

Elementare Werte werden mit dem ==-Operator verglichen. Es können Variablen beliebigen Typs verglichen werden. Der Operator achtet nur auf die Bits.

if (a == b) { …} betrachtet die Bits in a und b und gibt true zurück, wenn das Bitmuster gleich ist (der Operator kümmert sich allerdings nicht um die Größe der Variablen, fehlende Nullen auf der linken Seite werden zur Not bei einer impliziten Typumwandlung ergänzt).

```
int a = 3;

byte b = 3;

if (a == b) { // true }
```

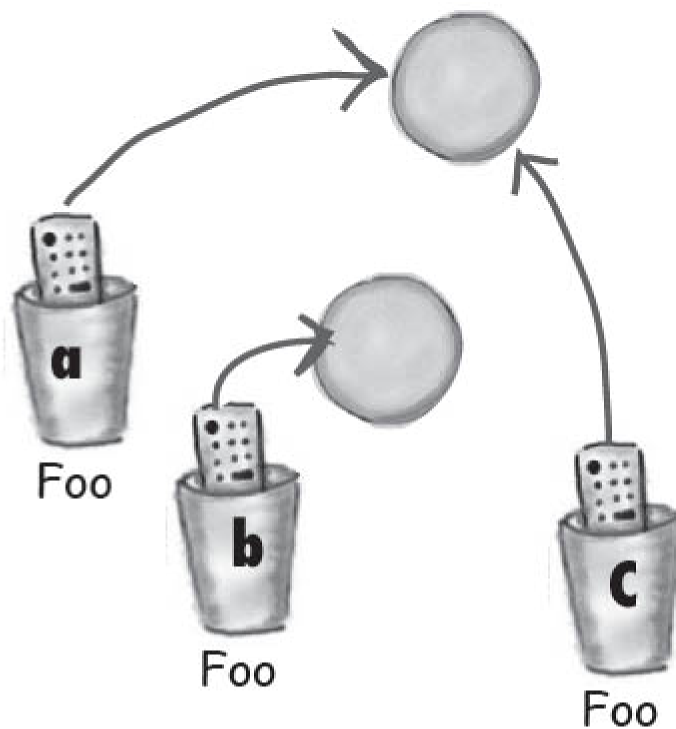


Um zu überprüfen, ob zwei Referenzen gleich sind (sich auf das gleiche Objekt auf dem Heap beziehen), benutzen Sie ebenfalls den ==-Operator

Vergessen Sie nicht, dass den ==-Operator nur das Bitmuster in der Variablen interessiert. Die Regeln sind die gleichen – ob es sich nun um eine elementare oder eine Referenzvariable handelt. Das heißt, der ==-Operator gibt true zurück, wenn zwei Referenzvariablen auf das gleiche Objekt verweisen! In diesem Fall kennen wir das Bitmuster nicht (weil es von der JVM abhängt und vor uns verborgen ist), aber wir *wissen*, dass es unabhängig von seinem Aussehen *für Referenzen auf dasselbe Objekt gleich ist*.

```
Foo a = new Foo();

Foo b = new Foo();
```

```
Foo c = a;

if (a == b) { } // false

if (a == c) { } // true

if (b == c) { } // false
```



PUNKT FÜR PUNKT

- Kapselung gibt Ihnen die Kontrolle darüber, wer die Daten in Ihrer Klasse auf welche Weise ändern darf.

- Markieren Sie eine Instanzvariable als *private*, damit sie nicht durch einen direkten Zugriff verändert werden kann.
- Erstellen Sie eine öffentliche (*public*) Mutator-Methode, also einen Setter, um zu kontrollieren, wie anderer Code mit Ihren Daten interagiert. Sie können einen Setter beispielsweise mit Validierungscode ausstatten, um sicherzustellen, dass der Wert nicht zu etwas Ungültigem verändert wurde.
- *Instanz*variablen erhalten Standardwerte, selbst wenn Sie nicht explizit einen Wert zuweisen.
- *Lokale*Variablen, zum Beispiel innerhalb von Methoden, erhalten keine Standardwerte. Sie müssen grundsätzlich initialisiert werden.
- Verwenden Sie ==, um zu testen, ob zwei elementare Variablen den gleichen Wert haben.
- Verwenden Sie ==, um zu testen, ob zwei Referenzvariablen gleich sind, das heißt auf das gleiche Objekt verweisen.
- Benutzen Sie .equals(), um zu sehen, ob zwei Objekte gleich sind (aber nicht unbedingt das gleiche Objekt), zum Beispiel um zu überprüfen, ob zwei String-Objekte die gleiche Zeichenfolge enthalten.

Spitzen Sie Ihren Bleistift

Was ist zulässig?

Welche der Methodenaufrufe auf der rechten Seite sind erlaubt, wenn wir die unten angegebene Methode voraussetzen?

Machen Sie ein Häkchen neben die erlaubten Zeilen. (Einige Anweisungen dienen dazu, die in den Methodenaufrufen verwendeten Werte zuzuweisen.)

```
int calcArea(int height, int width) {

   return height * width;

}

int a = calcArea(7, 12);

short c = 7;

calcArea(c, 15);

int d = calcArea(57);

calcArea(2, 3);

long t = 42;

int f = calcArea(t, 17);

int g = calcArea();
```

```
calcArea();

byte h = calcArea(4, 20);

int j = calcArea(2, 3, 5);
```

Antworten auf Seite 93.

SEIEN Sie der Compiler

Jede der Java-Dateien auf dieser Seite steht für eine vollständige Quelldatei. Ihre Aufgabe ist es, herauszufinden, ob diese Dateien kompiliert werden. Falls nicht, wie würden Sie sie reparieren, falls ja, was wären ihre Ausgaben?

A

```
class XCopy {
  public static void main(String[] args) {
    int orig = 42;
    XCopy x = new XCopy();
    int y = x.go(orig);
    System.out.println(orig + " " + y);
  }
  int go(int arg) {
    arg = arg * 2;
    return arg;
  }
}
```

B

```
class Clock {
  String time;
  void setTime(String t) {
```

```
    time = t;
  }
  void getTime() {
    return time;
  }
}
class ClockTestDrive {
  public static void main(String[] args) {
    Clock c = new Clock();
    c.setTime("1245");
    String tod = c.getTime();
    System.out.println("time: "+tod);
  }
}
```

→ **Antworten auf Seite 93.**

Ein ganzer Pulk von Java-Komponenten spielt in voller Kostümierung das Partyspiel »Wer bin ich?«. Und Sie sollen anhand der Tipps, die sie Ihnen geben, herausfinden, um wen es sich jeweils handelt. Gehen Sie davon aus, dass sie immer die Wahrheit über sich selbst erzählen. Falls etwas gesagt wird, das auf mehrere von ihnen zutreffen könnte, schreiben Sie alle auf, auf die der Satz passt. Tragen Sie den Namen eines oder mehrerer Anwesender in die leeren Zeilen neben dem Satz ein.

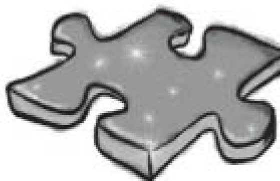
Die heutigen Gäste:

Instanzvariable, Argument, return, Getter, Setter, Kapselung, public, private, Pass-by-Value, Methode

Wer bin ich?

Eine Klasse kann eine beliebige Anzahl davon haben.	______________
Eine Methode kann nur eins davon haben.	______________
Dies kann implizit umgewandelt werden.	______________
Ich ziehe es vor, wenn meine Instanzvariablen privat sind.	______________
Eigentlich bedeutet es »eine Kopie erstellen«.	______________
Nur Setter können diese aktualisieren.	______________
Davon kann eine Methode mehrere haben.	______________
Ich gebe per definitionem etwas zurück.	______________
Ich sollte nicht mit Instanzvariablen verwendet werden.	______________
Ich kann viele Argumente haben.	______________
Ich übernehme per definitionem ein Argument.	______________
Sie helfen bei der Umsetzung der Verkapselung.	______________
Ich fliege immer solo.	______________

→ Antworten auf Seite 93.



Vermischte Nachrichten

Rechts sehen Sie ein kurzes Java-Programm. Zwei Blöcke des Programms fehlen. Ihre Herausforderung besteht darin, **die möglichen Codeblöcke** (unten links) **den Ausgaben zuzuordnen**, die Sie sähen, wenn die Blöcke eingefügt würden.

Einige Ausgabezeilen werden möglicherweise mehrmals benutzt. Verbinden Sie die Codeblock-Kandidaten mit den passenden Ausgaben auf der Kommandozeile.

Kandidaten:

```
i < 9

index < 5
```

```
i < 20

index < 5
```

```
i < 7

index < 7
```

```
i < 19

index < 1
```

Mögliche Ausgabe:

```
14 7
```

```
9 5
```

```
19 1
```

```
14 1
```

```
25 1
```

```
7 7
```

```
20 1
```

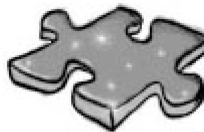
```
20 5
```

```
public class Mix4 {

  int counter = 0;

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

```
    int count = 0;

    Mix4[] mixes = new Mix4[20];

    int i = 0;

    while (          ) {

      mixes[i] = new Mix4();

      mixes[i].counter = mixes[i].counter + 1;

      count = count + 1;

      count = count + mixes[i].maybeNew(i);

      i = i + 1;

    }

    System.out.println(count + " " +

                       mixes[1].counter);

}

  public int maybeNew(int index) {

    if (              ) {

      Mix4 mix = new Mix4();
```

```
      mix.counter = mix.counter + 1;

      return 1;

    }

    return 0;

  }

}
```

Antworten auf Seite 94.



Pool-Puzzle

Ihre ***Aufgabe*** ist es, die Codeschnipsel aus dem Pool zu fischen und auf den Leerzeilen im Code zu platzieren. Der gleiche Codeschnipsel darf **nicht** mehrmals verwendet werden, und Sie werden nicht alle Schnipsel brauchen. Ihr ***Ziel*** ist es, eine Klasse zu erstellen, die kompiliert wird und bei der Ausführung die gezeigten Ausgaben erzeugt.

Antworten auf Seite 94.

Ausgabe

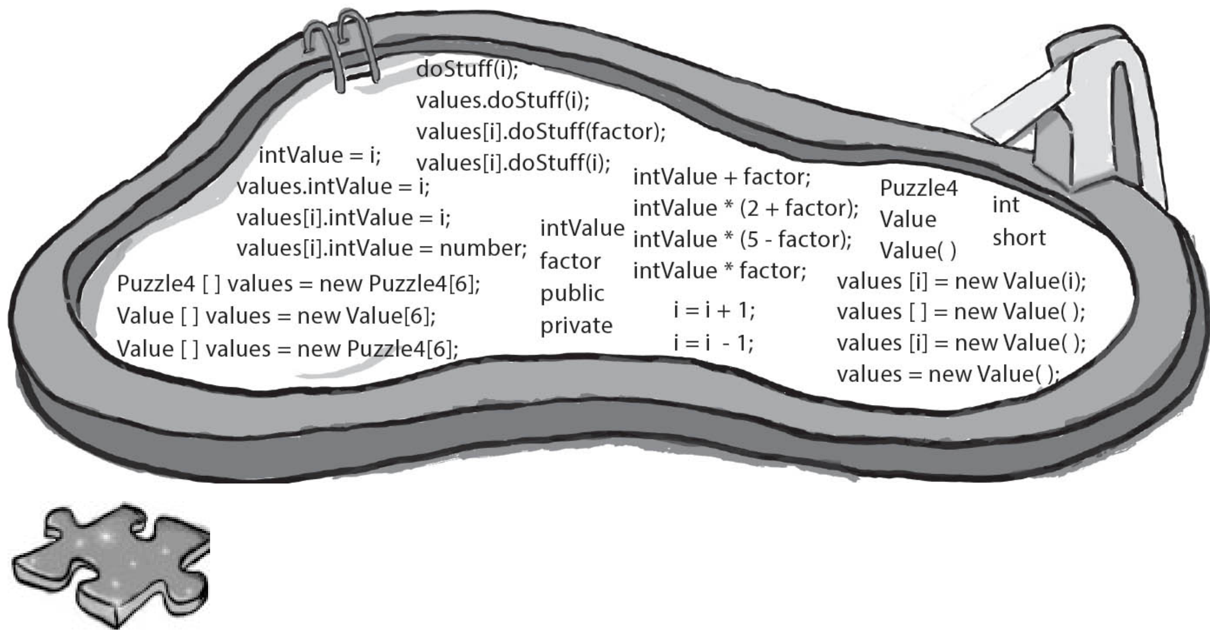
Datei Bearbeiten Fenster Hilfe Bauchklatscher

```
%java Puzzle4
result 543345
```

```
public class Puzzle4 {

  public static void main(String [] args) {

    ___________________________________

    int number = 1;

    int i = 0;

    while (i < 6) {

      ___________________________

      ___________________________

      number = number * 10;

      _________________

    }

    int result = 0;

    i = 6;

    while (i > 0) {

      _________________

      result = result + ___________________

    }
```

```
    System.out.println("result " + result);

  }

}

class ___________ {

  int intValue;

  ________ ______ doStuff(int _________) {

    if (intValue > 100) {

      return _________________________

    } else {

      return _________________________

    }

  }

}
```

Hinweis: Jeder Codeschnipsel im Pool darf nur einmal verwendet werden!



Unruhige Zeiten in Stim-City

Jai erstarrte, als Buchanan die Knarre in Jais Rippen rammte. Er wusste, dass Buchanan genauso dumm wie hässlich war, und wollte das Riesenbaby nicht erschrecken. Buchanan dirigierte Jai in das Büro seines Bosses. Aber Jai hatte keine Fehler gemacht (in letzter Zeit zumindest) und dachte, dass ein kleines Schwätzchen mit Buchanans Boss Leveler nicht so problematisch werden könnte. In den letzten Wochen hatte er massenhaft Neuro-Stimmer an die Westküste verschoben. Eigentlich sollte Leveler ganz zufrieden mit ihm sein. Schwarzmarkt-Stimmer waren immer noch die beste Geldpumpe weit und breit. Und außerdem waren sie absolut harmlos. Die meisten Stim-Junkies, die ihm über den Weg gelaufen waren, stiegen nach einer Weile aus und kehrten ins Leben zurück. Na, vielleicht waren sie etwas weniger bei der Sache als vorher.

Kurzkrimi

Levelers »Büro« war eine verbeulte Blechbüchse mit Lenkrad, aber nachdem Buchanan ihn hineinbugsiert hatte, sah Jai, dass einige Modifikationen vorgenommen worden waren, um das Etwas mehr an Speed und das Waffenarsenal einzubauen, von dem lokale Bosse wie Leveler träumten. »Jai, mein Junge«, zischte Leveler, »ein Vergnügen, dich mal wieder zu sehen.« »Ganz meinerseits. Denk ich zumindest …«, sagte Jai, der die Drohung in Levelers Begrüßung fühlte. »Bei uns sollte alles im Lot sein, Leveler? Oder ist mir da irgendetwas entgangen?« »Ha! Du sorgst dafür, dass alles ziemlich gut aussieht,

Jai, dein Umsatz stimmt. Aber in letzter Zeit ist mir, sagen wir, ein kleiner ›Einbruch‹ aufgefallen …«, sagte Leveler.

Jai zuckte unwillkürlich zusammen. Früher war er ein Spitzen-Hacker gewesen. Aber jedes Mal, wenn jemand herausfand, wie man die Sicherheit eines Systems aufbrechen kann, richtete sich eine unerwünschte Aufmerksamkeit auf ihn. »Kann ich nicht gewesen sein, Mann«, sagte Jai, »ist die Nachteile nicht wert. Hab dem Hacken den Rücken gekehrt. Ich verkauf mein Zeug und kümmere mich um meine eigenen Angelegenheiten.« »Ja, ja«, lachte Leveler, »ich bin sicher, dass du bei der Sache sauber bist. Aber bis dieser neue Hacker ausgesperrt ist, habe ich bereits große Summen verloren!« »Na dann viel Glück, Leveler, vielleicht setzt ihr mich einfach hier ab, und ich mach mich auf und bring noch ein paar ›Einheiten‹ für dich unter die Leute, bevor ich mich für heute zurückziehe«, sagte Jai schnell.

»Ich fürchte, so einfach ist das nicht, Jai. Buchanan hier sagt mir, es hieße, dass du im Augenblick auf Java 37.3.2 NE bist«, insistierte Leveler. »Neural-Edition? Klar, ich spiel etwas herum. Und?«, antwortete Jai mit einem leicht mulmigen Gefühl in der Magengegend. »Über Neural-Edition lass ich die Stim-Junkies wissen, wo es die nächste Lieferung gibt«, erklärte Leveler. »Das Problem ist, dass ein paar Stim-Junkies so lang draufblieben, bis sie herausgefunden hatten, wie man sich in meine Lagerhaltungsdatenbank hackt.« »Ich brauch jemanden, der schnell denken kann, jemanden wie dich, Jai, der einen Blick auf meine StimDrop-Java-NE-Klasse wirft – Methoden, Instanzvariablen und den ganzen Salat – und herausfindet, wie sie hereinkommen. Sollte …« »HE!«, stöhnte Buchanan. »Ich will nicht, dass Hacker-Abschaum wie der in meinem Code herumschnüffelt!« »Ganz ruhig, großer Junge!« Jai sah seine Chance. »Ich bin sicher, du hast mit deinen Zugriffsmodi ganze Arbeit geleistet …« »Was willst du Bit-Fummler mir eigentlich erzählen?«, rief Buchanan. »Ich hab alle Methoden auf Junkie-Level öffentlich gemacht, damit sie auf die Drop-Site-Daten zugreifen können, aber alle kritischen Lagerhaltungsmethoden privat gelassen. Keiner von draußen kann auf diese Methoden zugreifen, Kollega!«

»Ich glaub, ich kann dein Leck aufspüren, Leveler, wollen wir Buchanan nicht hier an der Ecke absetzen und eine kleine Rundfahrt um den Block machen?«, schlug Jai vor. Buchanan griff nach seiner Knarre, aber Levelers Schießeisen saß bereits in Buchanans Nacken. »Gib auf, Buchanan«, sagte Leveler verächtlich, »lass die Knarre fallen und steig aus. Ich glaube, Jai und ich müssen ein paar Pläne machen.«

Welchen Verdacht hat Jai?

Wird er Levelers Wagen mit heilen Knochen verlassen?

→ **Antworten auf Seite 94.**

Lösungen zu den Übungen

Spitzen Sie Ihren Bleistift (von Seite 87)

```
int a = calcArea(7, 12);

short c = 7;

calcArea(c, 15);
```

```
int d = calcArea(57);

calcArea(2, 3);
```

```
long t = 42;

int f = calcArea(t, 17);

int g = calcArea();
```

```
calcArea();

byte h = calcArea(4, 20);

int j = calcArea(2, 3, 5);
```

SEIEN Sie der Compiler (von Seite 88)

A

Die Klasse XCopy lässt sich so kompilieren und ausführen! Die Ausgabe ist: 42 84. Denken Sie daran, dass Java mit Pass-by-Value arbeitet (was heißt, dass die Kopien der Werte übergeben werden), die Variable orig wird von der Methode go( ) nicht verändert.

```
class Clock {

  String time;

  void setTime(String t) {

    time = t;

  }
```

B

```
String getTime() {
    return time;
  }
}
```

Hinweis: Getter-Methoden haben per definitionem einen Rückgabewert.

```
class ClockTestDrive {
  public static void main(String[] args) {
    Clock c = new Clock();
    c.setTime("1245");
    String tod = c.getTime();
    System.out.println("time: " + tod);
  }
}
```

Wer bin ich? (von Seite 89)

Eine Klasse kann eine beliebige Anzahl davon haben.	`Instanzvariablen, Getter, Setter, Methode`
Eine Methode kann nur eins davon haben.	`return`
Dies kann implizit umgewandelt werden.	`return, Argument`
Ich ziehe es vor, wenn meine Instanzvariablen privat sind.	`Kapselung`
Eigentlich bedeutet es »eine Kopie erstellen«,	`Pass-by-Value`
Nur Setter können diese aktualisieren.	`Instanzvariablen`

Davon kann eine Methode mehrere haben.	**`Argument`**
Ich gebe per definitionem etwas zurück.	**`Getter`**
Ich sollte nicht mit Instanzvariablen verwendet werden.	**`public`**
Ich kann viele Argumente haben.	**`Methode`**
Ich übernehme per definitionem ein Argument.	**`Setter`**
Sie helfen bei der Umsetzung der Verkapselung.	**`Getter, Setter, public, private`**
Ich fliege immer solo.	**`return`**

Puzzle-Lösungen

Pool-Puzzle (von Seite 91)

```
public class Puzzle4 {

  public static void main(String[] args) {

    Value[] values = new Value[6];

    int number = 1;

    int i = 0;

    while (i < 6) {

      values[i] = new Value();
```

```
      values[i].intValue = number;
      number = number * 10;
      i = i + 1;
    }
    int result = 0;
    i = 6;
    while (i > 0) {
      i = i - 1;
      result = result + values[i].doStuff(i);
    }
    System.out.println("result " + result);
  }
}
class Value {
  int intValue;
  public int doStuff(int factor) {
    if (intValue > 100) {
```

```
        return intValue * factor;
      } else {
        return intValue * (5 - factor);
      }
    }
  }
```

Ausgabe:

```
Datei  Bearbeiten  Fenster  Hilfe  Bauchklatscher
%java Puzzle4
result 543345
```

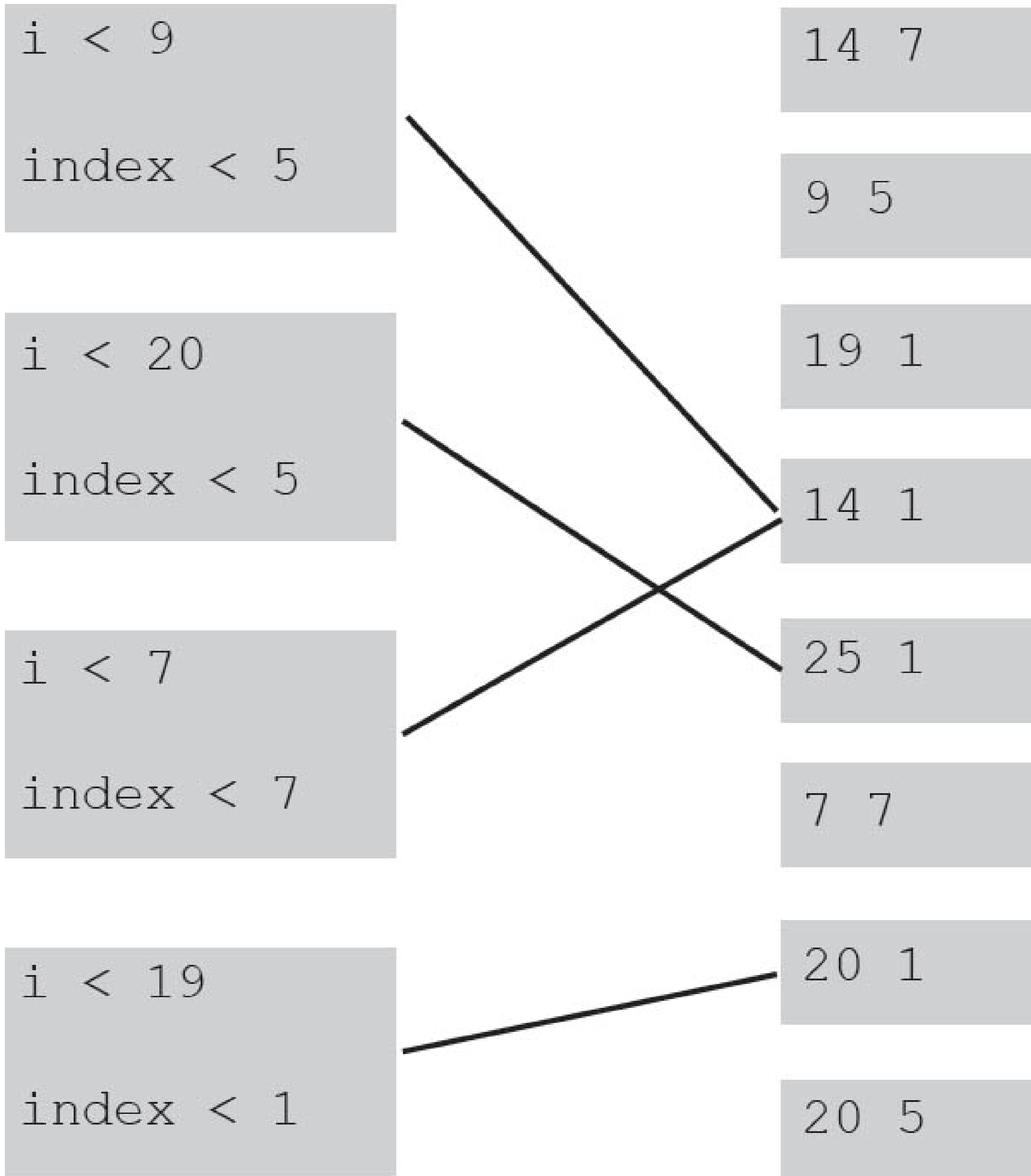
Kurzkrimi (von Seite 92)

Welchen Verdacht hat Jai?

Jai wusste, dass Buchanan nicht unbedingt die hellste Birne in der Lampe war. Als Buchanan über seinen Code sprach, sagte er kein Wort über seine Instanzvariablen. Jai hatte den Verdacht, dass Buchanan zwar seine Methoden richtig behandelt, aber versäumt hatte, seine Instanzvariablen als private zu markieren. Dieses Versäumnis konnte Leveler schnell Tausende gekostet haben.

Vermischte

Nachrichten (von Seite 90)



Index

Symbole

!= und ! (ungleich) 151

% (Prozentzeichen) in Formatierungsstring 297–300

&&, || (boolesche Operatoren) 151

++ — (Inkrement/Dekrement) 106, 115

-> (Pfeiloperator) für Lambda-Ausdrücke 388

. (Punktoperator) 36, 54, 61, 80

// (Kommentarsyntax) 12

:: (Methodenreferenz) 408

< (Kleiner-als-Operator) 13

<=, ==,!=, >, >= (Vergleichsoperatoren) 13, 86, 151, 348

<> (Diamant-Operator) 312–313, 698

= (Zuweisungsoperator) 13

== (Gleichheitsoperator) 13, 86, 348

> (Größer-als-Operator) 13

{} (geschweifte Klammern) für Klassen und Methoden 12

2-D-Grafiken zeichnen 471, 472–475, 501–503

A

Abfragen

- optionale Rückgabewerte 410–414
- Optionen für abschließende Operationen 410–412
- Stream-Pipelines als Abfragen auf Collections 380, 385

abschließende Operationen 377

eifrig (eager) 382–383
in Stream-Operationen 379
Optionen für Collection-Abfragen 410–412
stapeln 380
abstrakte Klassen 202–209
Bedingungen für Verwendung 229
Interface-Implementierung 226
Konstruktoren in 253
und Polymorphie 199, 208–209
und statische Methoden 278–282
abstrakte Methoden 205–206, 222, 226, 396
accept() 601
ActionEvent 467–468, 481–482, 523
ActionListener 481, 482, 491
actionPerformed() 467, 481, 482
add(anObject), ArrayList 137
addActionListener() 467–469
Akzessoren und Mutatoren *siehe* Getter und Setter
alphabetische (natürliche) Sortierfolge in Java 315
and- und or-Operatoren (&&, ||) 151
Animation 492–495
Annotationen 692
Anschluss-Streams 543
Anweisungen 12
Argumente 74, 76
an super() übergeben 257
Autoboxing 292
catch 428

Getter und Setter 79
mit generischen Typen 358–362
polymorph 189–190, 192
und no-arg-Konstruktoren 247–248
Zahlenformatierung 297, 302
Argumentliste 248–250, 302
ArrayList 132–139, 314
als generische Klasse 322–323
Autoboxing 291
Beziehung zu List im Sortierprojekt 310
Casting 231
Diamant-Operator 312–313
HashSet anstelle von 347
Referenzen vom Typ Object enthaltend 213–215
StartupBust-Objekt 142
Typparameter mit 323
und Werte von generischen Typen 320, 321
Arrays 19, 59–62
Elemente als Objekte 59
für die Deklaration mehrerer Rückgabewerte 78
Objektverhalten in 83
polymorph 188
Punktoperator in 61
SimpleStartupGame, Bug reparieren 128–130
Vergleich mit ArrayList 137–140
atomare Transaktionen, Nebenläufigkeit von Threads 646
atomare Variablen 655–657
AtomicInteger 656

Attribute 30

aufgegebene Objekte *siehe* Garbage Collections

äußere Klasse, Beziehung zu innerer Klasse 484–486

Autoboxing von elementaren Typen 291–292

Autocloseable 577

awaitTermination 629

B

BeatBox-App 422

- Chatclient und -server 588–589
- Clientprogramm 674–681
- Drum-Pattern speichern 579–582
- GUI für 528–533
- MIDI-Musikplayer 423
- Objekte speichern 540
- Serverprogramm 681–682

bedingte Verzweigungen 15

Bedingungsausdrücke 13, 15

Benennung

- Klassen und Interfaces 154–156
- Variablen 50–52, 53, 61

Bereiche, BorderLayout 514–517

Bierlied-Applikation 16

Bilder im GUI-Widget 471, 473

BindException 593

Blockade (Deadlock), Synchronisierung 654

boolean, Elementartyp 51, 53

boolesche Ausdrücke 13

Autoboxing mit 292
ungleich (!= und !) 151
Variablen, inkompatibel mit Integer-Werten 14
Werte aus einer Collection abfragen 410
boolesches anyMatch 377
BorderLayout, Layoutmanager 478–482, 511, 513, 514–517
BoxLayout, Layoutmanager 513, 521
break-Anweisungen 105
Buffer 565, 566
BufferedReader 566, 594
BufferedWriter 565, 572
Buttons, GUI 463–468
ActionEvent 481–482
BorderLayout 514–517
FlowLayout 519–520
innere Klasse, Zwei-Button-Code 487
Reaktion und Reaktionszeiten 464–465
byte, Elementartyp 51, 53
Bytecode 2

C

C, Programmiersprache 56
CAS-(Vergleichen-und-Austauschen-)Operationen 655–656
Casting 78, 111, 117, 218, 551
catch-Argument 428
catch-Blöcke *siehe* try/catch-Blöcke
char, Elementartyp 51, 53
Chatclient, App 588–589, 604–608, 632–633

Chatserver, App 588–589, 606–608
check.addItemListener(this) 526
Checkbox (JCheckBox) 526
checkUserGuess() 145
checkYourself() 102, 104, 130
Clientapplikationen, Netzwerk 588–600, 604–608, 632–633
Client/Server-Beziehung 589–593
Codeküche
- BeatBox-App 674–682
- BeatBox-Muster abspeichern 579–582
- einfacher Chatserver 606
- GameHelper-Klasse 152–153
- GUI für BeatBox 528–533
- Musik mit Grafiken 496–503
- Songs-Klasse 398–399
- Töne abspielen 445–453

collect() 377, 378, 410
Collection-API 345–346, 376
Collections, Klasse 314, 323
Collections.sort() 314–315
- Comparator 331–338
- compare() 333
- und List.sort() 332
- vom Typ Object anstelle von String 317–319

Collections
- ArrayList *siehe* ArrayList
- Factory-Methoden 356–357
- generische Typen für Typsicherheit 320–324

häufige Operationen 374
List *siehe* List, Interface
Map 355, 409
parametrisierte Typen 137
Streams als Abfragen 385, 387
und Nebenläufigkeit 662–666
verbesserte for-Schleife 116
Zähloperationen auf 410
Collectors, Klasse 378, 383, 387, 409
Collectors.joining 409
Collectors.toList 387, 409
Collectors.toMap 409
Collectors.toSet 409
Collectors.toUnmodifiableList 356, 387, 409
Collectors.toUnmodifiableMap 409
Collectors.toUnmodifiableSet 409
Comparable-Interface 354
Collections.sort() 327–330
Vergleich mit Comparator 332, 335
Comparator-Interface
als SAM-Typ 397
Lambda-Ausdrücke mit 341–343, 390–394
sortieren 314, 331–338
und Methodenreferenz 408
und TreeSet 354
Vergleich mit Comparable 332, 335
compare() 332, 333, 341
compareAndSet method 655–656

compareTo() 329–330, 332, 352
Compiler 2, 10–11, 687
Compiler-sichere Dokumentation, Annotationen als 692
computeIfAbsent 693
computeIfPresent 694
ConcurrentModificationException 663
Container (Hintergrundkomponenten), GUI 510, 511
contains() 403
ControllerEvent 497, 500
Convenience-Methoden 357–358, 387
CopyOnWriteArrayList 665–666
CountDownLatch 625

D

-d (Verzeichnis), Flag 687
DailyAdviceServer 602
Dateien, Quellcodedateien, Struktur 7
Datenformatierung 302
Datenstrukturen *siehe* Collections
DDD (Deadly Diamond of Death) 225
Deklarationen
    Exceptions 426, 441–443
    Methode 78, 144, 205–206, 222, 238
    Objekt 186–188
    Variable 50–52, 54, 84, 85, 116, 144, 238
Deserialisierung 551–557
dezimal (%d), Formatierungstyp 301
Diamant-Operator (<>) 312–313, 698

distinct(), Stream 375, 406–407
Domainnamen, umgekehrt, als Package-Namen 685
doppelte Ergebnisse, entfernen 406–407
double, Elementartyp 51, 53
Duck-Konstruktor 243–248, 250–251, 281–283
duplizierter Code, Vermeidung durch Vererbung 184

E

E-Lernkarten, Beispiel, in Textdatei speichern 560
einen Wert auspacken (Wrapper) 290
Elementartypen 49, 51
    als reservierte Wörter 53
    Bitgröße 241
    deklarieren 50–52
    Größen für Variablen 51
    in Arrays 59
    Objekte speichern 545
    Objekte vergleichen 86
    Wrapper 290–294
Enums 696–697
equals() 86, 349–351, 697
    erstellen 379–381
    map-Operation 405–407
Event-Handling 465–471
    Grafiken erhalten 477
    JCheckBox 526
    JList 527
    Listener-Interface 466–469

MidiEvent 449–452, 497–503
statische Methoden 498–499
Event-Objekt 470
Event-Quelle 467–469
Exception-Handling 421, 426–444
finally-Block 433, 444, 574–575
Flusskontrolle 432–433
mehrfache Exceptions 438
try-with-resources-Anweisung 576–577
try/catch-Blöcke *siehe* try/catch-Blöcke
Exceptions
abfangen *siehe* try/catch-Blöcke
auslösen 429–432
ausweichen 441–443
BindException 593
deklarieren 436, 441–443
mehrfach 435
Methoden 425
Nebenläufigkeit und Collections 663
NumberFormatException 294
polymorph 436
überprüft, Vergleich mit Laufzeit 430
Executors 626
Executors, Klasse 615
ExecutorService 615, 626–629
ExecutorService.shutdown() 615, 629
ExecutorService.shutdownNow() 629
explizites Casting 78

extends, Schlüsselwort 328–330

Extreme Programming 101

F

Factory-Methoden 356–358, 387, 615

Farbverlauf, Grafikobjekt 475

File, Klasse 564

File, Objekt 564

FileInputStream 551

FileOutputStream 542, 543

FileReader 566

Files, Klasse 572–573

FileWriter 559, 565

Filter-Streams *siehe* verkettete Streams

filter() 400–403

final, Schlüsselwort

- Klassen 191, 285, 286
- Methoden 191, 285
- Variablen 275, 284–286
- zu Felddeklaration hinzufügen 660, 666

finally-Block 433, 444, 574–575

findFirst(), optional 377

float (Fließkommazahl) (%f), Formatierungstyp 301

float, Elementartyp 51, 53

FlowLayout, Layoutmanager 513, 518–520

Flussdiagramm für Startups versenken 97

Flusskontrolle, Exceptions 432–434

for-Schleifen 114–116

SimpleStartup-Klasse 105
verbessert 106, 116
Vergleich mit forEach() 370–373
forEach() 370, 388, 393, 694
Fork-Join, Framework 695
format() 298
Format-Spezifizierer 297–298, 300–302
Formatierung, Zahlen 296–302
Formatter, Klasse 296
Frames, GUI 462, 511, 522
Friesen, Jeff, Java I/O, NIO and NIO.2 597
Function, in map-Methode 405
funktionales Interface, Lambda-Ausdruck 389–396

G

GameHelper-Klasse 112, 142, 152–153
GameHelper-Objekt 143
Garbage Collections 40, 57–58, 262–265
Garbage Collectible Heap 238
Geltungsbereich, Variable 260–267
Generics 320–324
extends oder implements 328–330
Klassen 321–323
Methoden 324, 375
Typparameter 362
und polymorphe Argumente 358–362
geschweifte Klammern ({}), für Klassen und Methoden 12
getPreferredSize() 522

getSequencer() 426

Getter und Setter 79–82, 646

getUserInput() 112

Gleichheit 348, 349

Gleichheitsoperator (==) 13, 86, 348

Gradle 686

Grafik 471–475. *Siehe auch* GUI

Graphics, Superklasse 474

Größer-als-Operator (>) 13

GUI 461–501

    abstrakte Klassen in 204

    BeatBox-App 528–533, 674

    BorderLayout 478–482, 511

    BoxLayout 513

    Buttons *siehe* Buttons, GUI

    Event-Handling 465–471

    FlowLayout 513

    Grafiken erstellen 471–475

    innere Klassen 484–494

    Komponenten 462, 471–475, 510, 523–527

    Layoutmanager 511–522

    Listener-Interface 466–469

    Swing 462, 509–533

H

hashCode() 348–351

HAT-EIN-Test für Vererbung 179–183

Hauptcode 99

Heap 40, 57, 238–241
hexadezimal (%x), Formatierungstyp 301
Hintergrundkomponenten (Container), GUI 510, 511
Hitchens, Ron, Java NIO 597
HTML-API, Dokumentation 160

I

I/O 540, 571
- Daten in Textdatei speichern 559–571
- Deserialisierung 551–555
- Exception-Handling 574–577
- Netzwerke *siehe* Netzwerk
- Objekte speichern 541–558
- Serialisierung 541–550, 554–557
- Streams 543

if-Anweisung 15, 697–698
if, Test 15
if/else-Anweisung 12, 15
Immutabilität 688, 700
immutable Objekte 658–661, 665–666
import-Anweisung 155, 157
Importe, statisch 303
increment(), synchronisieren 652
Indexposition, List 345
InetSocketAddress 591
Initialisierung
- Instanzvariablen 84
- mit Konstruktoren 246–248

statische Variablen 283
Inkrement/Dekrement-Operatoren (++ und —) 106, 115
innere Klassen 191, 337, 484–494
2-D-Grafiken zeichnen 501
Beziehung zu äußerer Klasse 484–486
Lambda-Ausdrücke mit 490–491
Zwei-Button-Code 487
InputStreamReader 596
Instanzen, innere und äußere Klasse 485–486
Instanzvariablen 34, 35
auf dem Heap 241
auf Parametertypen abbilden 76
deklarieren 84, 238
Fehlen in Math-Klasse 276
Getter und Setter 79
in Serialisierungsprozess 544–546
initialisieren 84
Lebensdauer und Geltungsbereich 260–267
Pass-by-Value/Pass-by-Copy 77
private Zugriffsmodifier 81
Setter-Methoden für 80–82
Standardwerte 84
Tier-Simulationsprogramm 173, 174
transient 549, 550, 553
Unmöglichkeit der Verwendung mit statischen Methoden 279
Unterklasse 169
Vergleich mit lokalen Variablen 85, 238–240
Vergleich mit statischen Variablen 304–305

int, Array-Variable 59

int, Elementartyp 51, 53

Integer-Werte, Inkompatibilität mit booleschen Variablen 14

interaktive Komponenten, GUI 510

interface, Schlüsselwort 226–231

Interfaces 199

  Benennung 154–156

  funktionales Interface 389–396, 491

  Polymorphie 226–231

IST-EIN-Test 179–180, 183, 188, 253

Iterationsausdruck, for-Schleifen 114

Iterationsvariable deklarieren, verbesserte for-Schleife 116

J

Java-API 125–164

  Dokumentation 158–162

  Klassen und Packages 154–162. *Siehe auch* Packages, Java-API

Java Collections, Framework 309. *Siehe auch* Collections

Java I/O, NIO and NIO.2 (Friesen) 597

Java NIO (Hitchens) 597

Java Virtual Machine *siehe* JVM

Java-fähiges Haus 17

Java-Modulsystem 161

java.nio.channels, Package 597

java.util, API 314

JavaFX 464

JavaSound, API 421, 423

Java

Codestruktur 7–8
einrichten
Funktionsweise 2–3
Geschichte 4
Geschwindigkeit und Speicherverbrauch 4
Grundelemente 41
Versionen, Namenskonventionen 5
JCheckBox 526
JComponent, Klasse 510
JFrame 462, 510, 522
JPanel 510, 518
JPanel.paintComponent() 472–473, 495
JPEG in Widget 473
JScrollPane 524
JShell 684
JTextArea 524–525
JTextArea.requestFocus() 524
JTextArea.selectAll() 524
JTextArea.setLineWrap(true) 524
JTextArea.setText() 524
JTextField 523
JTextField.requestFocus() 523
JTextField.selectAll() 523
JVM (Java Virtual Machine) 2, 9–11

K

Kamingespräche
for-Schleife und forEach-Methode, Vergleich 371

Instanzvariablen und statische Variablen, Vergleich 304–305
Rollen von JVM und Compiler 10–11
Variable, Diskussion über Leben und Tod 266–267
Kanäle, Client- und-Server-Netzwerke 587
lesen/empfangen mit 594–596
SocketChannel 591, 594, 595, 596–602
kanonischer Konstruktor *siehe* Konstruktoren
Klassen 8, 28, 41, 72. *Siehe auch* Vererbung
abstrakt *siehe* abstrakte Klassen
Bedingungen für Erstellung 229
Bestandteile 7
Deserialisierung und Versionierung 556–557
Dokumentation 162
Effizienz des Nicht-Speicherns mit Objekt 553
entwerfen 34
Executors 615
File 564
Files 572–573
finale 285, 286
für immutable Daten 658, 661
generisch 321–323, 328–330
Graphics2D 474–475
Hierarchiedesign mit Vererbung 175
in Objekterstellung 36–37
innere *siehe* innere Klassen
Instanzvariablen deklarieren, innerhalb 238
JComponent 510
konkret 202–212

Lambda-Ausdrücke als 389

Math 276–280

mehrere Interfaces implementieren 228

mit main() 9

Object-Superklasse 210–217

ohne Typparameter 324

Paths 572–573

PetShop-(Tierhandlung-)Programm, Änderungen am Klassenbaum 221–228

Random 111

Records 699–700

Sequencer 424–427

SimpleStartupGame 99–124

Tester-Klasse 36

Thread 609–610

Tier-(Animal-)Simulationsprogramm 173, 174

und Java-API-Packages 154–162, 685, 686

und Konstruktoren 243

und statische Variablen 283

und Typen 50

Vergleich mit Objekten 35

vollständige Namen in Java-Bibliothek 155–157

Klassen erweitern *siehe* Vererbung

Kleiner-als-Operator (<) 13

Kommandozeilenargumente, MidiEvent 452

Kommata, für Formatierung großer Zahlen 296, 298

Kommentarsyntax (//) 12

kompakter Konstruktor 700

konkrete Klassen 202–212

Konstanten 41, 284, 696
Konstruktoren 243–259
Aufgabe von 244
in Deserialisierung 552
Initialisierung von Objektzustand 245–248
privat 191, 251, 278, 282
Rückblick 251
Superklasse 252–259
überladen 258–259
überschreiben 700
verketten 253–259
Kurzschlussoperatoren (&& ,||) 151

L

Lambda-Ausdrücke 340–346, 388–397
Anatomie 391–392
Aufruf der Single Abstract Method 389
forEach-Methode 370
innere Klasse ersetzen durch 490–491
map-Operation 405, 408, 693–694
Methodenreferenz als Ersatz 408
Parameter 393–394
Predicate implementieren 402
Threads 612–614, 621
void als (Nicht-)Rückgabewert 393
latch.countDown 625
Laufzeit-Exceptions, Vergleich mit überprüften Exceptions 430
Layoutmanager 509, 511–522, 526

BorderLayout 478–482, 513–517
BoxLayout 513
eines Frames ändern 522
FlowLayout 513, 518–520
unterschiedliche Entscheidungsregeln 512
limit(), Stream 375–377, 381
List, Interface 345
Comparator mit 331
nicht modifizierbare Ausgaben 356–357, 387
sammeln in 383, 400, 409
sortieren mit 310–319
Verwendung, Vergleich mit Implementierungstyp 313
list.addListSelectionListener(this) 527
List.of() 357
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION) 527
list.setVisibleRowCount(4) 527
List.sort() 332
Listener 467–469, 481
ActionListener 482, 491
check.addItemListener(this) 526
list.addListSelectionListener(this) 527
Nicht-GUI-Event 497
Listener-Interface, Event-Handling 466–469
Literale, Werte zuweisen an 52
lokale Variablen
auf dem Stack 238
deklarieren 238
Leben und Geltungsbereich 260–261, 266–267

Parameter als 74
Referenzen auf Heap 240
Typinferenz 698
Vergleich mit Instanzvariablen 85, 238–240
long count() 377
long, Elementartyp 51, 53

M

main() 8, 12, 14, 27
Exceptions ausweichen 442
in Tester-Klasse 36
Klassen mit 9
Multithreading 611, 613
SimpleStartupGame, Klasse 108, 110–111
StartupBust, Objekt 142
Verwendung 38
makeEvent() 499–500
Map, Interface 355
Collection-API 346
Lambda-Ausdrücke 693–694
Schlüssel/Wert-Paare 345
und Collections 409
Map.of() 357
Map.ofEntries() 357
map(), Stream 375
Math, Klasse 276–280, 288–289
Math.abs() 288
Math.max() 289

Math.min() 289
Math.random() 111, 288
Math.round() 289
Math.sqrt() 289
Maven, Java-Projektstruktur 686
Mehrfachvererbung, Problem 224–225
Message, MidiEvent 449–451, 498
Metadaten 572
Methoden 7, 8, 30, 34. *Siehe auch* lokale Variablen
- abstrakt 205–206, 222, 226, 396
- Argumente *siehe* Argumente
- Aufruf von nicht statischen aus statischen Methoden 280
- Autoboxing 292
- beschreibender Name, beste Vorgehensweise 698
- deklarieren oder ausweichen 441–443
- deklarieren und implementieren 144–145
- Exception-Handling 425–426, 443
- final 191, 285
- Funktionsweise mit Vererbung 177–178
- für gesamten Code verfügbar machen 41
- generisch 321, 324, 362
- Gleichheit von Variablen- und Parametertypen 76
- in Stack 238, 239
- Listener-Interface 466
- lokale Variablen deklarieren innerhalb 85, 238
- Math, Klasse 288–289
- mehrere Parameter mit 76
- mehrere Rückgabewerte, Deklaration 78

nicht abstrakte Methoden mit abstrakten Klassen 206
Parameter *siehe* Parameter
Rückgabewerte von 75
Signatur 206
SimpleStartup-Klasse, Testcode für 102–104
statisch 276–280
Superklasse 230
Testcode 101
Tier-Simulationsprogramm 173
überladen 193
überschreiben *siehe* Methoden überschreiben
und Rolle des Typs Object im Code 212, 215
und statische Variablen 286
Unterklassen 182
Varargs 691
Vererbbarkeit 178
Vergleich mit Konstruktoren 245
Methoden überschreiben 32, 169–194
hashCode() und equals() 350–351
Object, Klasse 212
Regeln, um Vertrag einzuhalten 192
Superklasse 169, 230
toString() 316
Methodenreferenz, Stream 408
MIDI-Musikplayer 423–424, 446–453
MidiEvents 449–452, 497–503
Mockups 310
Modifier, für abstrakte Klassen 202

MovieTestDrive-Klasse 37

Multithreading 609–630, 695. *Siehe auch* Nebenläufigkeit

parallele Streams 695

Runnable, Interface 612–617

Scheduling 617–619

SimpleChatClient, vollständig 632–633

sleep() 622–624

Stack 610–614

Thread-Pools 626–629

Thread, Zustände 616

Threads koordinieren 622–625

Musikvideo 496–503

Mutatoren und Akzessoren *siehe* Getter und Setter

N

natürliche Sortierfolge in Java (alphabetisch) 315

Nebenläufigkeit 639–669

atomare Variablen 655–657

Collections 662–666

immutable Objekte 658–661

Multithreading 630

Race Condition 641–643

sperren 645

Synchronisierung 646–654

verlorene Aktualisierung, Szenario 650–653

Vor- und Nachteile 666

Netzwerk 587–635

Client und Server, Beziehung 589–593

einfache Chatclient-App 604–608
einfache Chatserver-App 606–608
Kanäle *siehe* Kanäle
nextInt() 111
Nicht-Kurzschluss-Operatoren (& , |) 151
nicht öffentliche Klasse 191
NIO (nicht blockierende I/O) 561
NIO.2 597
no-arg-Konstruktoren 247–248, 250
null-Referenz 58, 265
NumberFormatException 294
numerische Elementartypen 51

O

Object, Klasse 210–217
ObjectInputStream 551
ObjectOutputStream 542, 543
ObjectOutputStream.close() 542
ObjectOutputStream.writeObject() 542
Objekt-Graph 546, 548, 550
Objekte neu zeichnen, GUI 492–495
Objekte. *Siehe auch* Arrays; Strings
Array-Elemente als 60, 83
auf dem Heap 238–240
deklarieren 186–188
erstellen 36–37, 55, 242–254
für Garbage Collection qualifiziert 262–265
Gleichheit 86, 348, 349

immutabel 658–661, 665–666
instanziieren 103
Instanzvariablen innerhalb von 241
Lambda-Ausdrücke als 389
Lebenszyklus 253, 260–267
sperren 645–649, 656
Superklassenkonstruktoren 252–259
und Collections 374
Vergleich mit Klassen 35
Verhalten 539. *Siehe auch* Klassen
vom Typ Object 212
Zustand speichern 539, 541–558
Zuweisung 55, 186–188
Objektreferenzen *siehe* Referenzvariablen
Objektverhalten *siehe* Methoden
Objektzustände *siehe* Instanzvariablen
OO (objektorientierte) Entwicklung 14, 27–48. *Siehe auch* Klassen; Objekte
Event-Handling 481
Objektzustand speichern 539
Vererbung 168–185
Operatoren. *Siehe auch* Elementartypen
and- und or-Operatoren 151
Inkrement/Dekrement (++ und —) 106, 115
ist gleich (==) 13, 86, 348
Kurzschluss 151
Nicht-Kurzschluss 151
Postinkrement 105
und Autoboxing 293

Vergleich 13, 86, 151, 348
optimistisches Sperren 656
optionaler Wert, Rückgabe aus Collection-Abfrage 410–414
OutputStream 596

P

Packages, Java-API 154–155
Code organisieren 686
Klassen platzieren in 686
kompilieren und ausführen 687
Konflikte zwischen Klassennamen verhindern 685
umgekehrte Domainnamen als Namen für Packages 685
Verzeichnisstruktur 686
Panel, GUI 511–522
parallele Streams 695
parallelStream 695
Parameter. *Siehe auch* Argumente
auf Typen von Instanzvariablen abbilden 76
Lambda-Ausdrücke 402
Typ *siehe* Typparameter
und lokale Variablen 85
und Methoden 74, 76
parametrisierte Typen 137
Parsing-Methoden 294
Pass-by-Value/Pass-by-Copy 77
Path, Interface 572–573
Paths, Klasse 572–573
PetShop-Programm 220–228

Phras-O-Mat, Code 18–19

Pipelines, Stream *siehe* Stream-Pipeline

Polymorphie

- abstrakte Klassen 202–205, 208–209
- Argumente und Rückgabetypen 189–190
- Graphics, Superklasse 474
- Interface-Implementierung 226–231
- Methoden 205–206
- mit generischen Typen 321, 358–362
- Object, Klasse 210–215
- Referenz- und Objekttypen unterschiedlich 188–189
- und Exceptions 436–440
- und List, Vergleich mit ArrayList 313

Postinkrement-Operator 105

Predicate (Kriterium) 375, 402

primitive Datentypen *siehe* Elementartypen

print, Vergleich mit println 15

print() 595

printf() 296

println() 595

printStackTrace() 429

PrintWriter 595, 596, 602

private Konstruktoren 191, 251, 278, 282

private, Zugriffsmodifier 81, 689

protected, Zugriffsebene 689, 690

protected, Zugriffsmodifier 689

Prozentzeichen (%) in Formatstring 297–300

Pseudocode *siehe* Vorcode

public, Zugriffsmodifier 81–82, 689

Punktoperator (.) 36, 54, 61, 80

Q

Quellcodedateien, Struktur 7

Quelle (source), Event 466

QuizCardBuilder 560–563

QuizCardPlayer 567–569

R

Race Conditions 630, 650

Random, Klasse 111

random() 111

Ratespiel, Beispiel 38–40

Ratgeberonkel 598–599

Rätsel

- GUI-Kreuzworträtsel 536
- Heap-Chaos 66
- Java-Kreuzworträtsel 22, 120, 164, 456, 536
- Kurzkrimi 67, 92, 270–271, 415, 669
- Pool-Puzzle 24, 44, 91, 196, 234, 416, 506
- Vermischte Nachrichten 90, 121

Read-Eval-Print-Loop *siehe* REPL

Reader 594

Records 699–700

Referenzvariablen 49

- auf Heap 240
- auf null setzen 265
- Casting 218

Entblößung durch Punktoperator vermeiden 80
Garbage Collection Heap 57–58
Gleichheit 348
Größe 56–57
in Arrays 61–62
in Arrays, Zugriff auf 83
Lebenszyklus und Geltungsbereich 260–267
Methodenaufruf über 215
null-Referenz 58
Polymorphie 187–188
und Objekte vom Typ Object 213–215, 217
vergleichen 86
Von Kopf bis Fuß-Interview 56
zugewiesene Speichermenge 241
Zuweisung 264
Remote-Interface *siehe* RMI
REPL (Read-Eval-Print-Loop) 684
replaceAll 694
reservierte Wörter 53, 328
return, Schlüsselwort, Lambda-Ausdruck 390
RMI (Remote Method Invocation) 553
Rückgabetypen 75
Methoden überladen 193
polymorph 189–190, 192
Wert ignorieren 78
Rückgabewerte, Autoboxing mit 292
run() 612, 613
Runnable, Interface 612–617

runnable, Thread-Zustand 616
RuntimeExceptions 430
Ryan und Monica, Nebenläufigkeitsszenario 641–643, 646, 655–656

S

SAM (Single Abstract Method) 341, 389, 394
Schleifen 12, 13
  for 105, 106, 114–116, 370–373
  while 13, 115, 566
Schleifenblock 13
Schlüssel/Wert-Paare, Map 355
Schlüsselwörter in Java 53, 328
Scrolling (JScrollPane) 524, 527
Semikolon (;) 12
Sequencer-Klasse 424–427, 444, 446–447
Sequenz 446–447
Serialisierung 540, 541–550
  Objekt per Stream in Datei speichern 542–543
  Prozess 544–546
  Spielfiguren-Beispiel 554–555
  Versionierung 556–557
Serializable, Interface 547–550
serialVersionUID 557
Serverapplikation, Netzwerke 601–603
Server-Client-Beziehung 589–593
Server, Socket 601
ServerSocketChannel 601–602
Set, Interface 345, 349

Set.of() 357
setLayout(null) 522
setLocationCells() 102
Setter-Methoden 80–82
Setter *siehe* Getter und Setter
short, Elementartyp 51, 53
ShortMessage, Instanz 450
ShortMessage.setMessage() 450, 498
shutdown() 629
Sicherheit
  durch Package-Organisation in Java-Bibliothek 156
  und finale Klassen 191
SimpleStartupGame 98–124
  Klasse 108–111, 126–130
SimpleStartupTestDrive, Klasse 103–106
Single Abstract Method (einzelne abstrakte Methode) *siehe* SAM
skip(), Stream 375
sleep() 622–624
Socket 596–602
SocketAddress 594, 595
SocketChannel 591, 594, 595, 601–602
Song, Klasse 316
Song, Objekt 316–319
Songs-Klasse, Mockup für 311
sort() 318–319, 325–330
sorted() 375, 381, 390
sortieren
  Comparable, Interface 325–329

List 311–313
mit Comparator 331–338
TreeSet 352–354
Speicher
Garbage Collection 262–265
Stack und Heap im Objekt-Lebenszyklus 238–241
String-Immutabilität 688
sperren 645–649, 656
split() 570
Stack 238–241
Methoden aufrufen von 239
Superklassenkonstruktor 254–256
Threads 610–614
Variablendeklarationen 238
Stack-Frame 239
Stack-Variablen *siehe* lokale Variablen
Standardzugriffsebene 689, 690
Standardmethode 396
Standardwert
Instanzvariable 84
statische Variable 283
Startup-Klasse 138–139, 141, 150
Startup-Objekte 143
StartupBust-Klasse 141–148
Startups versenken, Spiel 96–97, 140–153
statische finale Variablen 275, 284, 696
statische Hilfsmethode 408
statische Importe 303

statische Methoden 396
Event-Handling 498–499
und Objektsperrung 649
und Wrapper 294
statische Variablen
initialisieren 283
und nicht statische Methoden 286
und Serialisierung 553
Vergleich mit Instanzvariablen 304–305
statischer Initialisierer 284
Stream-Pipeline
Streams (I/O)
in Serialisierung von Objekten 542–543
Nachrichten empfangen 594
Socket-Verbindungen 596
Textdateien lesen 566
Streams (Streams-API) 369, 373, 375–420
Ergebnisse erhalten von 378
Ergebnisse sammeln 409–410
filtern 400–403
Grundbausteine 379–381
Lambda-Ausdrücke 388–397
parallele Streams 695
stream() 376, 384
Wiederverwendung unmöglich 384
String-Arrays 19
String, Klasse 191
String.format(), Methode 296

StringBuilder 688
Strings 62
Daten in Textdatei speichern 559–571
ein- und auspacken (Wrapper) 294–295
Immutabilität 688
in Mockup-Songs-Klasse sortieren 311–313
Prozentzeichen (%) in 297–300
split() 570
wiederverwenden 688
Stuhlkriege-Szenario 28–35, 168–169
super, Schlüsselwort 182
super() 256, 258
Superklasse 31, 168–194
Beziehung zu Unterklassen 179, 184
Graphics 474
in Exception-Deklaration und catch 436–437
Referenztypen als 188
Superklassenversion einer Methode aufrufen 230
Überschreiben von Methoden aus 192
und keine Rückwärtsvererbung 182–186
und Mehrfachvererbung 224–225
Vertragsregeln 192
Superklassenkonstruktor 252–259
Swing 462, 464, 509–532
GUI für Beatbox 528–533
Komponenten 510, 523–527
Layoutmanager 511–522
switch, Anweisung 697–698

Synchronisierung für Nebenläufigkeit beim Objektzugriff 646–654
synchronized-Block oder -Methoden, Threads 647–649
synchronized, Schlüsselwort 646–649
Syntax 12, 14
System.out.print() 15
System.out.println() 15

T

TDD (Test-Driven Development) 101–103
temporär nicht lauffähige Threads 617–619
Test-Driven Development *siehe* TDD
Testbedingung, for-Schleife 114
Testcode 99, 145
Testcode, Annotationen in 692
Testklasse 36
Textbereich (JTextArea) 524–525
Textdatei
    Daten schreiben in 541, 559–564
    Daten speichern in 540
    lesen aus 566–571
Textfeld (JTextField) 523
this() 258
Thread (Ausführungsstrang), als allgemeiner Begriff 609–610
Thread-Pools 626–629
Thread-Scheduler 617–619
Thread, Klasse 609–610
Thread, Konstruktor 614
Threads *siehe* Multithreading

threadsichere Datenstruktur 664–666

throw-Klausel, Exceptions 426, 429–432, 436

Tier-Simulationsprogramm 172–178, 200–204

toString() 316

Track (Spur) 446

transient, Schlüsselwort 550

transiente Variablen 549, 553

TreeSet 352–354

try/catch-Blöcke 427–430

- Aufruf verpacken in 444
- Exception-Handling, Rolle in 443
- Flusskontrolle 432–433
- mehrere Exceptions abfangen 435, 438
- polymorph 436–440
- Reihenfolge bei mehreren 437–440

TWR-Anweisung (try-with-resources) 576–577

Typen

- für Lambda-Ausdruck 394
- Parameter und Methoden 78
- Variablen 50

Typinferenz 312, 698

Typmodifier, Zahlenformate 301

Typparameter

- ArrayList 323
- generische Methoden 324, 362
- nicht definiert in Klassendeklaration 324
- String in eckigen Klammern als 137

U

überladen
Konstruktoren 258–259
Methoden 193
überprüfte versus Laufzeit-Exceptions, Vergleich 430
überschreiben, Konstruktoren 700
Übungen
Beliebte Objekte 269
Codemagnete 20, 43, 64, 119, 163, 386, 455, 583, 634
Seien Sie … 21, 42, 63, 88, 118, 195, 306, 363, 395, 505
Spitzen Sie Ihren Bleistift 5–6, 15, 37, 52, 87, 107, 134–141, 145–147, 207, 250–251, 259, 287, 293, 334, 343, 353, 397, 431, 434, 440, 493, 502–503, 580, 600–601
Vermischte Nachrichten 23, 194, 372
Wahr oder falsch 307, 454, 582
Welches Layout? 534–535
Wer bin ich? 45, 89, 504, 631
Wer macht was? 374
Wie sieht das Bild aus? 232
Wie sieht die Deklaration aus? 233
ungleich (!= und !) 151
Unterklassen 31, 168–194
als Erweiterungen einer Superklasse 179–183
als Instanziierer unter abstrakten Klassen 203
Beziehung zu Superklasse 182
Erstellung, Bedingungen 229
Grenzen 191
Methodenimplementierungsdesigns für 174, 175
Schachtelungsebenen, beste Vorgehensweise 191
und Polymorphie 190

und Superklassenkonstruktoren 256

Unterstriche für große Zahlen, Formatierung 296

V

Varargs (variable Argumentlisten) 302, 691

Variablen

Benennung 50–52, 61

beschreibende Namen, beste Vorgehensweise 698

deklarieren 54, 84, 85, 116, 144

final 275, 284–286

in Arrays 59–62

Instanz *siehe* Instanzvariablen

konkrete Typen mit var 698

lokale *siehe* lokale Variablen

primitive *siehe* Elementartypen

Referenz *siehe* Referenzvariablen

statische *siehe* statische Variablen

Syntax 12

Typen 41

Typen vergleichen 86

var für lokale 698

von generischen Typen 321

Werte zuweisen an 52

verbesserte for-Schleife 106, 116

Verbindung, zwischen Client und Server 591–593

Vererbung 31, 168–185. *Siehe auch* Polymorphie

abstrakte Methoden, implementieren 206

Bäume, übertriebene Verschachtelung vermeiden 191

Beziehung zu Objekten 216

Ge- und Verbote 183

Subklassen *siehe* Unterklassen

Superklasse *siehe* Superklasse

Tier-Simulationsprogramm 172–178

Vorteile der Verwendung 184–185

Vererbungsbäume 191, 220–231

Vergleichen-und-Austauschen-Operationen *siehe* CAS

Vergleichsoperatoren 13, 86, 151, 348

Verkapselung 80–82

verkettete Objekte 19

verkettete Streams 379, 543, 571

verschütten, Werte von Variablen 52

Versions-ID, Serialisierung 556–557

vertikaler Scrollbalken 527

Vertrag

Klassenbaum modifizieren 220–226

öffentliche Methoden als 192–193, 219

Verzeichnisse

File-Objekt mit 564

Packages 573, 686

Verzweigungen (if/else) 12

virtueller Methodenaufruf 177

voll qualifizierte Namen, Package in Java-Bibliothek 155–157

Vorcode (prep code) 99

SimpleStartup-Klasse 100–101

StartupBust-Klasse 144–145

W

Werte, Variable
bei Methodenaufruf übergeben 74
ein- und auspacken (Wrapper) 290
Objektreferenz als 55
statische Variablen 283
while-Schleifen 13, 115, 566
Whitespace-Zeichen 12
Widgets 471
2-D-Grafiken zeichnen 471
Bilder auf 473
Wrapper-Konstruktor 290
Wrapper
für elementare Typen 290–294
Immutabilität 688
Optional, als 410–414
writeObject() 543
Writer 595

Z

Zahlen formatieren 296–302
Zähloperationen, auf Collections 410
Zeichen (%c) Formatierungstyp 301
Zufallszahlengeneratoren 19, 111
Zugriffsebenen 192, 193, 689–690
Zugriffskontrolle
Ebenen 689–690
Modifier 81–82, 689–690

- Objektsperren für 645, 647, 649, 656
- Polymorphie 192
- und Unterklassen 191

Zugriffsmodifier 81–82, 251, 689–690

Zuweisungsoperator (=) 13

Zuweisung
- Arrays 60
- Autoboxing 293
- Lambda-Ausdruck-Variablen 394
- Objekte 55, 186–188
- von elementaren Typen 52
- von Referenzvariablen 55

Zwischenoperationen 376
- »faule« Auswertung 383–384
- Operationen verketten 380
- Stream-Pipeline erstellen 379