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

Java™ von Kopf bis Fuß

Kathy Sierra
Bert Bates
Trisha Gee

Deutsche Übersetzung von
Jørgen W. Lang

O'REILLY®

Kathy Sierra, Bert Bates, Trisha Gee

Lektorat: Alexandra Follenius
Übersetzung: Jørgen W. Lang
Copy-Editing: Sibylle Feldmann, *www.richtiger-text.de*
Satz: Ulrich Borstelmann, *www.borstelmann.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Ellie Volckhausen, Susan Thompson, Michael Oréal, *www.oreal.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

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.

ISBN:
Print 978-3-96009-206-3
PDF 978-3-96010-748-4
ePub 978-3-96010-749-1
mobi 978-3-96010-750-7

2. Auflage 2023, Übersetzung der 3. englischen Auflage

Wieblinger Weg 17
69123 Heidelberg

Authorized German translation of the English edition of *Head First Java™ 3rd Edition*, ISBN 9781491910771 © 2022 Kathy Sierra, Bert Bates, and O'Reilly Media, Inc.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«.
O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt.
Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: *kommentar@oreilly.de*.

5 4 3 2 1 0

Von Kathy und Bert:

Für unsere Gehirne – dafür, dass sie immer da sind.

(trotz schwacher Beweise)

Von Trisha:

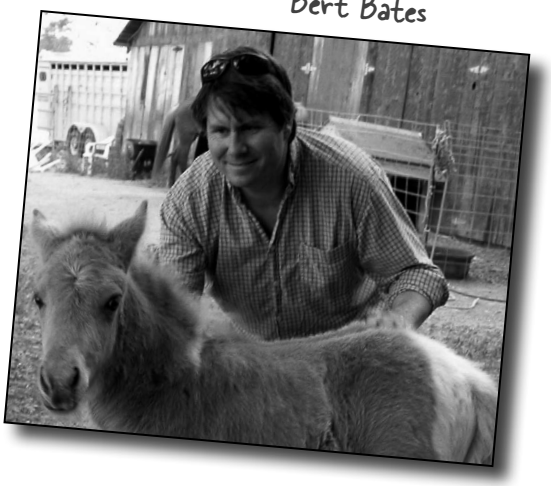
Für Isra – dafür, dass sie immer da ist.

(mit einer Menge an Beweisen)

Die Autorin und der Autor von Java von Kopf bis Fuß und die Schöpfer der Head-First-Buchreihe

Kathy Sierra

Bert Bates



Kathy interessiert sich für Lerntheorie seit ihrer Zeit als Spieleentwicklerin für Virgin, MGM und Amblin' und als Dozentin für New Media Authoring an der UCLA. Sie war Master-Java-Trainerin für Sun Microsystems, und sie gründete JavaRanch.com (jetzt CodeRanch.com), das in den Jahren 2003 und 2004 den Jolt Cola Productivity Award gewann. 2015 erhielt sie den Electronic Frontier Foundation's Pioneer Award für ihre Arbeit zur Schaffung kompetenter Nutzer und den Aufbau nachhaltiger Gemeinschaften.

In jüngster Zeit konzentriert sich Kathy auf modernste Bewegungswissenschaften und das Coaching zum Erwerb von Fähigkeiten, bekannt als Ecological Dynamics oder »Eco-D«. Ihre Arbeit, bei der sie Eco-D für das Training von Pferden einsetzt, führt zu einem weitaus humaneren Ansatz in der Reitkunst, was die einen erfreut (und die anderen, traurigerweise, verwirrt). Die Pferde, die das Glück haben, dass ihre Besitzer Kathys Ansatz anwenden, sind zufriedener, autonomer, gesünder und sportlicher als ihre traditionell trainierten Artgenossen.

Sie finden Kathy auf Instagram unter *@pantherflows*.

Bevor **Bert** Autor wurde, war er Entwickler, spezialisiert auf KI der alten Schule (hauptsächlich Expertensysteme), Echtzeit-Betriebssysteme und komplexe Planungssysteme.

Im Jahr 2003 schrieben Bert und Kathy *Java von Kopf bis Fuß* und starteten damit die Head-First-Reihe. Seitdem hat Bert weitere Java-Bücher geschrieben. Außerdem hat er Sun Microsystems und Oracle bei vielen ihrer Java-Zertifizierungen beraten. Heutzutage arbeitet er mit Coaches, Lehrern, Professorinnen, Autoren und Herausgeberinnen und hilft ihnen, den Lernenden ein noch besseres Training zu bieten.

Bert ist Go-Spieler und musste 2016 mit Schrecken (und Faszination) miterleben, wie AlphaGo Lee Sedol schlug. Neuerdings verwendet er Eco-D (Ecological Dynamics), um sein Golfspiel zu verbessern und seinen Papagei Bokeh zu trainieren.

Bert hat das große Glück, Trisha Gee seit mehr als acht Jahren zu kennen, und die Head-First-Buchreihe (Von Kopf bis Fuß) kann sich glücklich schätzen, Trisha zu ihren Autorinnen zu zählen.

Sie erreichen Bert per E-Mail unter *bertbates.hf@gmail.com*.

Co-Autorin von Java von Kopf bis Fuß, dritte englische Auflage

Trisha Gee

Trisha arbeitet bereits seit 1997 mit Java, und ihre Universität war vorausschauend genug, diese »strahlend neue« Sprache für den Informatikunterricht einzuführen. Seitdem arbeitet sie als Entwicklerin und Beraterin, erstellt Java-Applikationen für eine Reihe verschiedener Branchen, beispielsweise Bankenwesen, Fertigung und Non-Profit, sowie Low-Latency-Anwendungen für den Handel im Finanzsektor.

Trisha teilt leidenschaftlich gern das Wissen, das sie während ihrer Jahre als Entwicklerin auf die harte Tour erworben hat. Also wurde sie Developer Advocate, um eine Ausrede dafür zu haben, Blogposts zu schreiben, auf Konferenzen zu sprechen und Videos zu erstellen, mit denen sie einen Teil ihres Wissens weitergeben kann. Sie verbrachte fünf Jahre als Java Developer Advocate bei JetBrains und weitere zwei Jahre als Leiterin des JetBrains-Java-Advocacy-Teams. Während dieser Zeit lernte sie eine Menge über die Probleme, mit denen echte Java-Entwickler sich herumschlagen müssen.

In den vergangenen acht (!) Jahren hat Trisha immer mal wieder mit Bert darüber gesprochen, *Java von Kopf bis Fuß* endlich zu aktualisieren. Sie erinnert sich noch sehr gerne an die wöchentlichen Telefonate. Der regelmäßige Kontakt mit einem so kenntnisreichen und warmherzigen Menschen wie Bert half ihr, nicht verrückt zu werden. Berts und Kathys Ansatz, das Lernen zu fördern, bildet den Kern dessen, was sie in den vergangenen zehn Jahren zu erreichen versuchte.

Sie finden Trisha auf Twitter unter *@trisha_gee*.

Über den Übersetzer dieses Buchs

Jørgen W. Lang lebt und arbeitet als freier Autor (»CSS Kochbuch«) und Übersetzer in Oldenburg/Niedersachsen. Mitte der Neunzigerjahre des vergangenen Jahrhunderts begann er, sich mit dem damals noch jungen World Wide Web und seinen Möglichkeiten zu beschäftigen. Pünktlich zum Jahrtausendwendejahr erschien seine erste Übersetzung für den O'Reilly Verlag. Mittlerweile ist der Umfang seiner Übersetzungen auf mehr als 10.000 Seiten angewachsen.

Mit großer Energie und Ausdauer bringt Jørgen seit fast schon zwei Jahrzehnten Webseiten bei, das zu tun, was von ihnen erwartet wird – unabhängig davon, auf welchem Gerät sie betrachtet werden (elektrische Zahnbürsten ausgenommen).

Das zweite Standbein von Jørgen Lang ist die Musik. Außerhalb der Welt der semantischen Elemente, Selektoren und Objekte hat er sich einen Namen als hervorragender Gitarrist, Sänger, Komponist und Arrangeur gemacht und kann auf eine Vielzahl veröffentlichter Alben und mehrere Hundert Konzerte in aller Welt (z. B. für die UNESCO in Seoul) zurückblicken.

Der Inhalt (im Überblick)

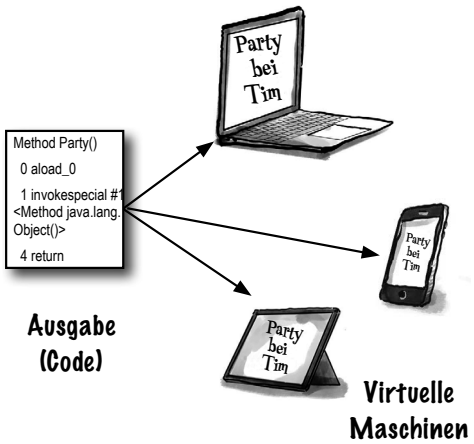
Der Inhalt (jetzt ausführlich)

Einführung

Ihr Gehirn und Java. Sie versuchen, etwas zu *lernen*, und Ihr *Hirn* tut sein Bestes, damit das Gelernte nicht *hängen bleibt*. Es denkt nämlich: »Wir sollten lieber ordentlich Platz für wichtigere Dinge lassen, z. B. für das Wissen, welche Tiere einem gefährlich werden könnten, oder dass es eine ganz schlechte Idee ist, nackt Snowboard zu fahren.« Tja, wie schaffen wir es nun, Ihr Gehirn davon zu überzeugen, dass Ihr Leben davon abhängt, etwas über Java zu wissen?

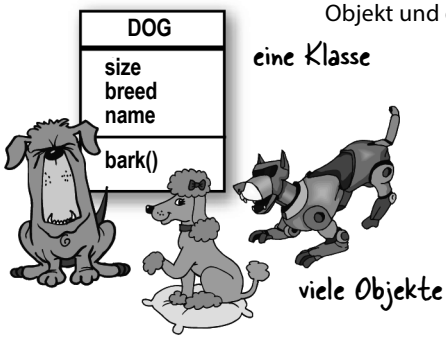
1 Die Oberfläche durchbrechen

Java bringt Sie an neue Orte. Seit seiner bescheidenen Veröffentlichung in der (schwächlichen) Version 1.02 hat Java Programmierer mit seiner freundlichen Syntax, seinen objektorientierten Features, der Speicherverwaltung und vor allem mit dem Versprechen auf Portabilität verführt. Wir machen mal eine kleine Kostprobe und schreiben ein bisschen Code, kompilieren ihn und lassen ihn laufen. Wir sprechen über die Syntax, Schleifen, Verzweigungen und alles, was Java so cool macht. Springen Sie rein!



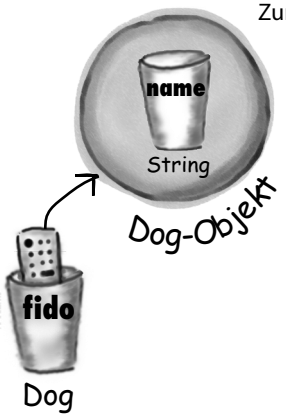
2 Die Reise nach Objectville

Ich dachte, hier gibt's Objekte. In Kapitel 1 haben wir den ganzen Code in die main()-Methode gepackt. Das ist jedoch nicht besonders objektorientiert – eigentlich überhaupt nicht. Aber jetzt ist endlich die Zeit gekommen, die prozedurale Welt hinter uns zu lassen. Nichts wie raus aus main() und ran an die Erstellung unserer eigenen Objekte! Wir werden erfahren, warum die objektorientierte Entwicklung in Java so viel Spaß macht. Außerdem werfen wir einen Blick auf den Unterschied zwischen einer Klasse und einem Objekt und darauf, wie Objekte Ihr Leben verbessern können.



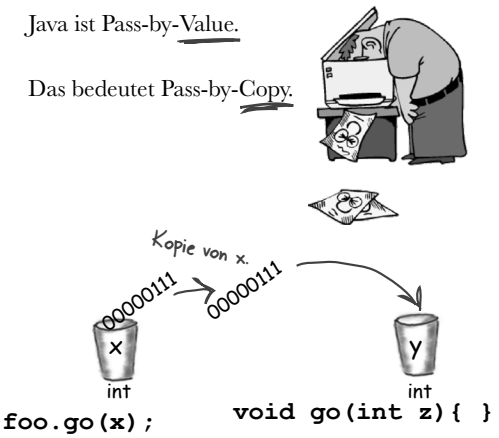
3 Verstehen Sie Ihre Variablen

Variablen gibt es in zwei Geschmacksrichtungen: elementare Typen und Referenztypen. Das Leben kann doch nicht nur aus Integer-Werten, Strings und Arrays bestehen! Wie wäre es mit einem HaustierBesitzer-Objekt und einer Instanzvariablen vom Typ Hund? Oder mit einem Auto-Objekt, das einen Motor hat? In diesem Kapitel werden wir die Geheimnisse der Java-Typen lüften und uns ansehen, was man alles als Variablen *deklarieren* kann, was Sie in einer Variablen *speichern* und mit einer Variablen *tun* können. Zum Schluss lernen Sie das Leben auf dem Garbage Collectible Heap hautnah kennen.



4 Wie sich Objekte verhalten

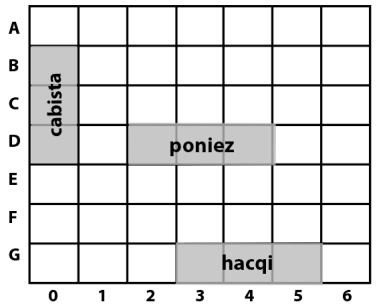
Zustand beeinflusst Verhalten, Verhalten beeinflusst Zustände. Wir wissen, dass Objekte über **Zustand** und **Verhalten** verfügen, die durch **Instanzvariablen** und **Methoden** repräsentiert werden. Nun sehen wir uns an, in welcher Beziehung Zustände und Verhalten zueinander stehen. Objekte haben ein Verhalten, das sich auf ihre Zustände 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.***



5 Methoden mit Superkräften

Jetzt wollen wir unsere Methoden mal etwas aufmöbeln. Wir haben uns mit Variablen beschäftigt, mit ein paar Objekten herumgespielt und ein bisschen Code geschrieben. Aber wir brauchen Werkzeuge. Wie **Operatoren**. Und **Schleifen**. Sie könnten nützlich sein, um **Zufallszahlen zu erzeugen**. Oder **einen String in einen int umzuwandeln**, ja, das wäre cool. Und warum lernen wir das alles nicht, indem wir sofort etwas Reales bauen? Dann sehen wir, wie es ist, ein Programm von Grund auf neu zu coden (und zu testen). **Vielleicht ein Spiel** wie Schiffe versenken.

Lasst uns ein Startups-versenken-Spiel bauen!



6 Die Java-Bibliothek verwenden

Java wird mit Hunderten vorgefertigter Klassen ausgeliefert. Sie müssen das Rad nicht neu erfinden, wenn Sie wissen, wie Sie das Benötigte in der Java-Bibliothek, der sogenannten **Java-API**, finden können. *Sie haben Besseres zu tun.* Wenn Sie Code schreiben, reicht es völlig, wenn Sie sich *nur auf die Teile* konzentrieren, die bei Ihrer Anwendung besonders sind. Die Java-Kernbibliothek ist ein gigantischer Haufen Klassen, die nur darauf warten, dass Sie sie als Bausteine für Ihre eigenen Programme verwenden.

»Gut zu wissen, dass das java.util-Package eine ArrayList-Klasse bereitstellt. Selbst hätte ich das aber nicht herausgefunden.«

- Julia, 31, Hand-Model

7 Besser leben in Objectville

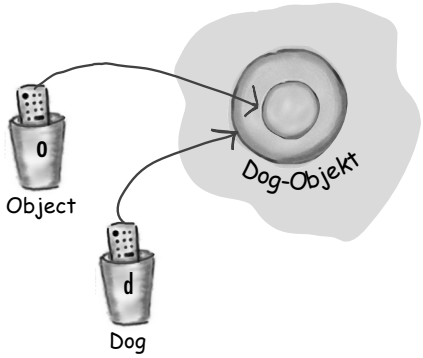
Planen Sie Ihre Programme mit der Zukunft im Blick. Was wäre, wenn Sie Code schreiben könnten, den ein *anderer* **problemlos** erweitern kann? Und was, wenn Ihr Code flexibel genug für diese lästigen Änderungen der Spezifikation in letzter Minute wäre? Wenn Sie auf den Polymorphie-Zug springen, lernen Sie die fünf Schritte zu besserem Klassendesign, die drei Tricks der Polymorphie und die acht Wege, Code flexibel zu machen. Dazu – wenn Sie sofort zugreifen – eine Bonusstunde mit den vier Tipps zur Nutzung von Vererbung inklusive.

8 Ernsthafte Polymorphie

Vererbung ist nur der Anfang. Um Polymorphie nutzen zu können, benötigen wir Interfaces. Wir müssen über einfache Vererbung hinausgehen und eine Stufe der Flexibilität erreichen, die man nur erhält, wenn man Schnittstellendefinitionen als Ausgangsbasis für den Entwurf und die Programmierung nimmt. Was eine Java-Schnittstelle ist? Eine zu 100 % abstrakte Klasse. Was eine abstrakte Klasse ist? Das ist eine Klasse, die nicht instanziiert werden kann. Und wozu soll die gut sein? Lesen Sie das Kapitel ...

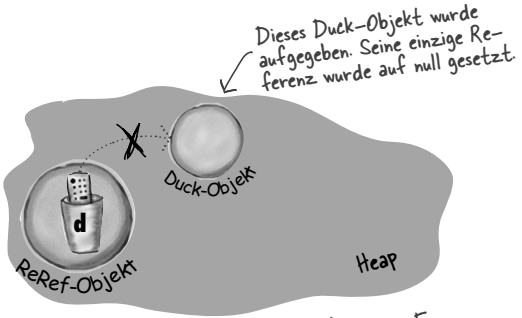
```
Object o = al.get(index);
Dog d = (Dog) o;
d.roam();
```

Das Object wieder zu dem Dog casten, das es die ganze Zeit war.



9 Leben und Sterben eines Objekts

Objekte werden geboren und Objekte sterben. Sie herrschen über das Leben eines Objekts. Sie entscheiden, wie und wann es *konstruiert* wird. Sie entscheiden, wann es zerstört wird. Der **Garbage Collector (gc)** will den Speicher wiederhaben. Wir sehen uns an, wie Objekte erzeugt werden, wo sie leben und wie man sie effizient gehen lässt. Das bedeutet, dass wir über den Heap, den Stack, Geltungsbereiche, Konstruktoren, Superklassenkonstruktoren, Nullreferenzen und mehr reden werden.



10 Zahlen, bitte!

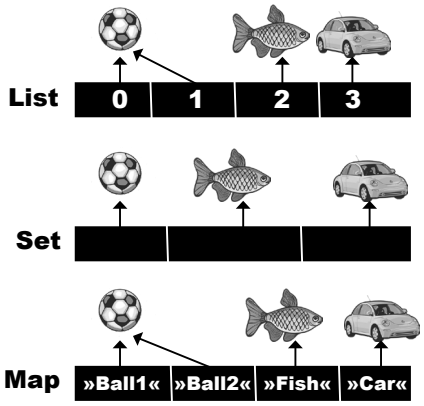
Rechnen Sie's aus. In der Java-API gibt es eine Menge praktischer Methoden für Absolutbeträge, zum Runden, zur Extremwertbestimmung etc. Aber wie sieht es mit der Formatierung aus? Vielleicht möchten Sie, dass Ihre Zahlen mit exakt zwei Nachkommastellen ausgegeben oder dass große Zahlen mit Tausenderpunkten unterteilt werden. Und wie steht es mit Datumsangaben? Und wenn Sie einen String in eine Zahl parsen wollen? Oder die Zahl in einen String? Als Erstes sehen wir uns an, was es für eine Variable oder eine Methode bedeutet, statisch zu sein.

Statische Variablen werden geteilt.

Alle Instanzen derselben Klasse teilen sich eine einzelne Kopie der statischen Variablen.

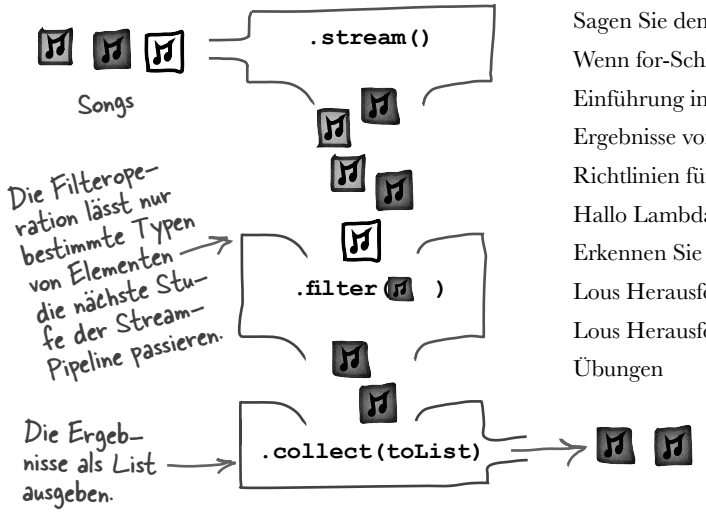
11 Datenstrukturen

Sortieren ist in Java ein Klacks. Ihnen stehen alle Werkzeuge für die Sammlung und Verarbeitung von Daten zur Verfügung, ohne dass Sie Ihren eigenen Sortieralgorithmus schreiben müssen. Im Java-Collections-Framework finden Sie Datenstrukturen für quasi jeden Zweck, den Sie sich nur vorstellen können. Brauchen Sie eine Liste, die einfach um neue Elemente erweitert werden kann? Wollen Sie etwas anhand seines Namens finden? Möchten Sie eine Liste erstellen, die automatisch Duplikate entfernt? Sie möchten Ihre Kollegen danach sortieren, wie oft sie Ihnen in den Rücken gefallen sind? Es ist alles da ...



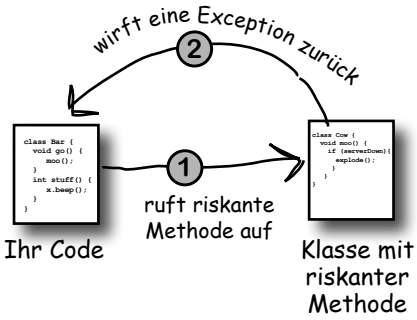
12 Lambdas und Streams: Was – nicht wie!

Was wäre, wenn ... Sie dem Computer nicht mitteilen müssten, WIE er etwas tun soll? In diesem Kapitel befassen wir uns mit der Streams-API, und Sie werden sehen, wie nützlich Lambda-Ausdrücke bei der Verwendung von Streams sein können. Außerdem werden Sie lernen, wie Sie mithilfe der Streams-API die Daten in einer Collection abfragen und umwandeln können.



13 Riskantes Verhalten

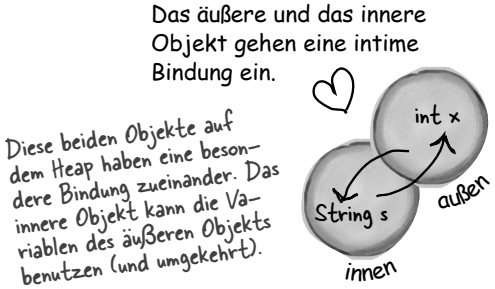
Dinge passieren. Die Datei fehlt. Der Server ist down. Auch wenn Sie ein noch so guter Programmierer sind – Sie können nicht alles unter Kontrolle haben. Wenn Sie eine riskante Methode schreiben, brauchen Sie Code, der mit den schlimmen Dingen umgeht, die passieren könnten. Aber woher wissen Sie, wann eine Methode riskant ist? Und wo platzieren Sie den Code, der die Ausnahmesituation behandelt? In diesem Kapitel programmieren wir einen MIDI-Musikplayer, der die riskante Java-Sound-API benutzt – daher sollten wir das besser alles schnell rausfinden!



14 Innen hui, außen GUI

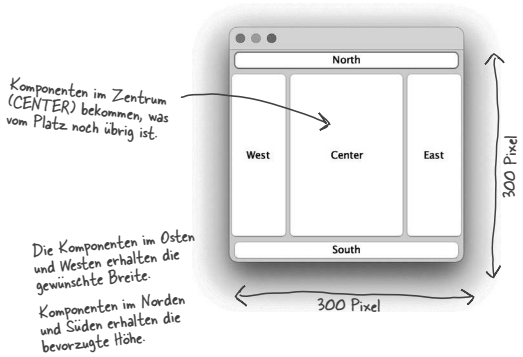
Sehen Sie den Tatsachen ins Auge: Früher oder später werden Sie GUIs erstellen müssen. Selbst wenn Sie davon ausgehen, dass Sie den Rest Ihres Lebens serverseitigen Code schreiben, werden Sie irgendwann Werkzeuge schreiben müssen, die ein GUI benötigen. Wir werden zwei Kapitel mit der Programmierung von GUIs zubringen und dabei grundlegende Features der Sprache Java kennenlernen, z. B. **Event-Handling** und **innere Klassen**. Wir stellen einen Button auf dem Bildschirm dar, zeigen ein JPEG-Bild an und versuchen uns sogar an einer kleinen Animation.

```
class MyOuter  {

    class MyInner {
       void go() {
         }
    }

}
```



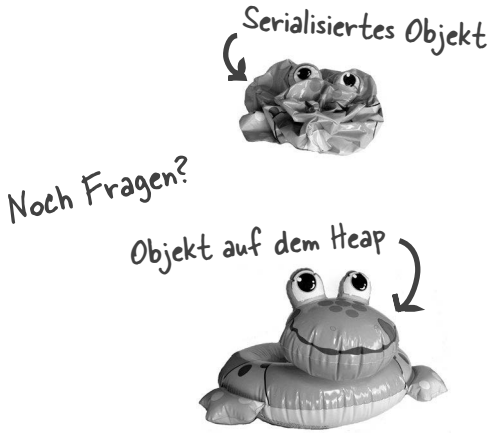
15 Mehr Schwung mit Swing

Swing ist einfach. Es sei denn, es ist Ihnen tatsächlich *wichtig*, wo die Dinge auf dem Bildschirm landen. Swing-Code *sieht* einfach aus, bis Sie ihn kompilieren und laufen lassen und denken: »Moment mal, *das* sollte aber *woandershin*!« Der Grund dafür, dass er *einfach* zu *programmieren*, aber *schwer* zu *kontrollieren* ist, ist der **Layoutmanager**. Mit ein bisschen Mühe bringen Sie Layoutmanager jedoch dazu, sich Ihrem Willen zu unterwerfen. In diesem Kapitel bringen wir unsere Swing-Künste in Schwung und lernen etwas über Widgets.



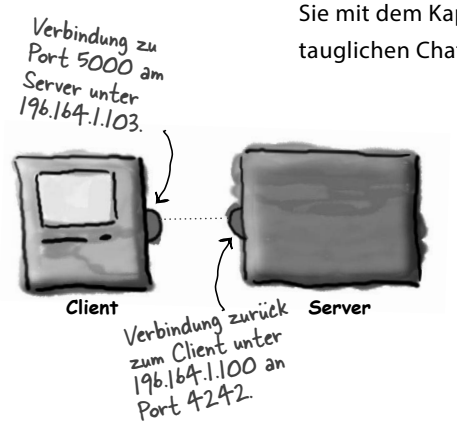
16 Objekte (und Text) speichern

Objekte lassen sich flach drücken und wieder aufblasen. Objekte haben einen Zustand und ein Verhalten. Das *Verhalten* steckt in der *Klasse*, aber der *Zustand* steckt in jedem einzelnen *Objekt*. Muss Ihr Programm Zustände speichern, können Sie das entweder *auf die harte Tour machen*, indem Sie jedes Objekt befragen und dann mühsam die Werte aller Instanzvariablen notieren. Oder **Sie gehen einfach nach OO-Art vor** – indem Sie lediglich das Objekt selbst gefriertrocknen (serialisieren) und deserialisieren, wenn Sie es zurückhaben wollen.



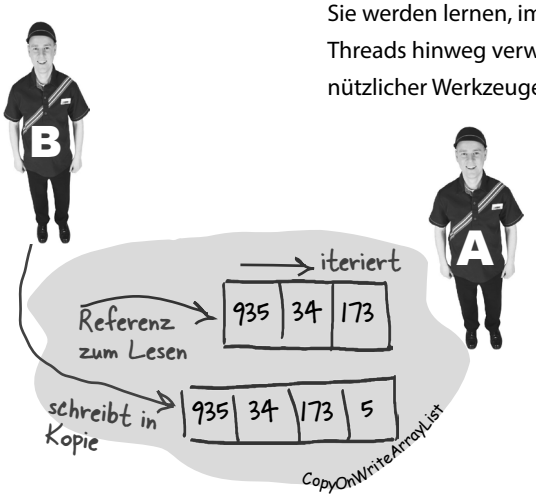
17 Eine Verbindung herstellen

Verbindung zur Außenwelt aufnehmen. Das ist nicht schwer. Um alle grundlegenden Netzwerkdetails kümmern sich Klassen in der java.net-Bibliothek. Einer der großen Vorteile von Java ist, dass das Senden und Empfangen über ein Netzwerk einfache Eingabe/Ausgabe ist, lediglich mit einem etwas anderen Anschluss-Stream am Ende der Kette. In diesem Kapitel erstellen wir *Client*-und *Server*-Sockets und *Clients* und *Server*. Bevor Sie mit dem Kapitel durch sind, haben Sie einen voll funktionsfähigen und Multithread-tauglichen Chatclient. Haben wir da gerade *Multithreading* erwähnt?



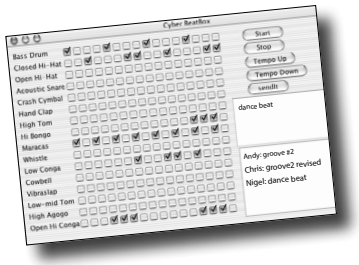
18 Nebenläufigkeitsprobleme behandeln

Es ist schwer, mehrere Dinge gleichzeitig zu tun. Multithreading-Code zu schreiben, ist einfach. Multithreading-Code zu schreiben, der wie erwartet funktioniert, kann viel schwieriger sein. In diesem letzten Kapitel zeigen wir Ihnen ein paar Dinge, die schiefgehen können, wenn mehrere Threads gleichzeitig arbeiten. Sie werden einige Werkzeuge in java.util.concurrent kennenlernen, die Ihnen helfen können, Multithreading-Code zu schreiben, der korrekt funktioniert. Sie werden lernen, immutable (unveränderliche) Objekte zu erstellen, die sicher über mehrere Threads hinweg verwendet werden können. Am Ende dieses Kapitels verfügen Sie über eine Reihe nützlicher Werkzeuge für die Arbeit mit Nebenläufigkeit.



A Anhang A

Die finale Codeküche. Der vollständige Code für unser BeatBox-Clientprogramm – Ihre Chance, ein Rockstar zu werden!



B Anhang B

Die (mehr als) zehn wichtigsten Themen, die es nicht ins Buch geschafft haben … Noch können wir Sie nicht ganz gehen lassen. Ein paar Sachen haben wir noch, aber dann sind Sie fertig. Diesmal meinen wir es ernst.

i Index

Einführung

In diesem Abschnitt beantworten wir die brennende Frage:
>>Und? Warum steht SO ETWAS in einem Buch über die Programmiersprache Java?<<

Für wen ist dieses Buch?

Wenn Sie ***alle*** folgenden Fragen mit »Ja« beantworten können:

1. **Haben Sie schon Programmiererfahrung?**
2. **Wollen Sie Java lernen?**
3. **Ziehen Sie eine anregende Unterhaltung beim Abendessen einer langweiligen technischen Vorlesung vor?**

dann ist dieses Buch etwas für Sie.

Dies ist KEIN Nachschlagewerk. *Java von Kopf bis Fuß* ist ein Buch zum *Lernen*, keine Enzyklopädie zu Java-Fakten.

Wer sollte eher die Finger von diesem Buch lassen?

Wenn Sie mindestens ***eine*** der folgenden Fragen mit »Ja« beantworten können:

1. **Beschränken sich Ihre Programmierkenntnisse auf HTML? Haben Sie keine Erfahrung mit Skriptsprachen?** (Wenn Sie schon mal etwas mit Schleifen und mit if-then-Logik gemacht haben, werden Sie mit diesem Buch prima zurechtkommen, aber HTML-Tags allein reichen vermutlich nicht.)
2. **Sind Sie ein Top-C++-Entwickler auf der Suche nach einem Nachschlagewerk?**
3. **Haben Sie Angst, etwas Neues auszuprobieren? Unterziehen Sie sich lieber einer Wurzelbehandlung, als Streifen mit Karos zu mischen? Glauben Sie, dass ein Technikbuch nicht ernst sein kann, wenn im Abschnitt zur Speicherverwaltung das Bild einer Ente auftaucht?**

dann ist dieses Buch ***nicht*** das Richtige für Sie.

[Anmerkung aus der Marketingabteilung: Wer hat den Abschnitt gestrichen, der darauf hinweist, dass dieses Buch etwas für jeden Besitzer einer gültigen Kreditkarte ist? Und was ist mit dieser >>Legen Sie Java auf den Gabentisch<<-Werbung? ... – Fred]

Wir wissen, was Sie denken.

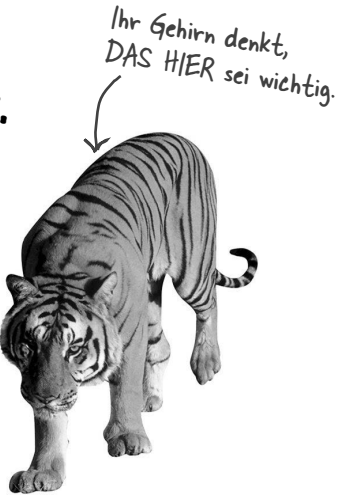
»Kann *das* wirklich ein ernsthaftes Java-Programmierbuch sein?«

»Was sollen die ganzen Abbildungen?«

»Kann ich auf diese Weise wirklich etwas *lernen*?«

»Und warum riecht es hier nach Pizza?«

Und wir wissen, was Ihr *Gehirn* gerade denkt.



Ihr Gehirn lechzt nach Neuem. Es ist ständig dabei, Ihre Umgebung abzusuchen, und es *wartet* auf Ungewöhnliches. So ist es nun einmal gebaut, und es hilft Ihnen zu überleben.

Heutzutage ist es eher unwahrscheinlich, dass Sie als Tigerfutter enden. Aber man weiß ja nie.

Also, was macht Ihr Gehirn mit all den normalen, langweiligen Routinesachen, die Ihnen so begegnen? Es tut *alles*, damit es dadurch nicht bei seiner *eigentlichen* Arbeit gestört wird: die *wirklich wichtigen* Dinge zu erfassen. Es gibt sich nicht damit ab, die langweiligen Sachen zu speichern, der »Das hier ist offensichtlich nicht wichtig«-Filter lässt sie gar nicht erst durch.

Aber woher *weiß* Ihr Gehirn, was wichtig ist? Angenommen, Sie machen eine längere Wildniswanderung und ein Tiger springt aus dem Gebüsch. Was passiert in Ihrem Kopf?

Neuronen feuern. Gefühle kochen hoch. *Chemische Substanzen durchfluten Sie*.

Und so weiß Ihr Gehirn:

Das muss wichtig sein! Vergiss es nicht!

Und jetzt stellen Sie sich vor, Sie sind zu Hause oder in der Bibliothek. Sie befinden sich in einem sicheren, warmen, tigerfreien Bereich. Sie studieren, bereiten sich auf eine Prüfung vor. Oder Sie versuchen, irgendein schwieriges technisches Thema zu lernen, von dem Ihr Chef glaubt, Sie bräuchten dafür eine Woche oder höchstens zehn Tage.

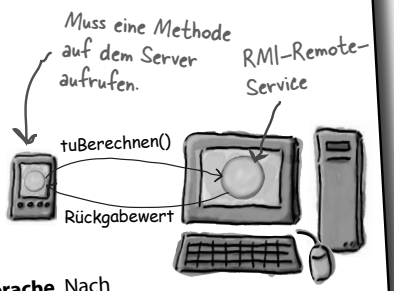
Da ist nur ein Problem. Ihr Gehirn versucht, Ihnen einen großen Gefallen zu tun. Es will sicherstellen, dass diese *offensichtlich* unwichtigen Inhalte keine knappen Ressourcen blockieren. Ressourcen, die besser dafür verwendet würden, die wirklich *wichtigen* Dinge zu speichern. Wie Tiger. Wie die Gefahren des Feuers. Oder dass Sie nie wieder in Shorts snowboarden sollten.

Leider gibt es kein Patentrezept, um Ihrem Gehirn zu sagen: »Hey, Gehirn, vielen Dank, aber egal, wie langweilig dieses Buch auch ist und wie klein der Ausschlag auf meiner emotionalen Richterskala gerade ist, ich will wirklich, dass du diesen Kram behältst.«

Wir stellen uns die Leserinnen und Leser dieses Buchs als aktive Lernende vor.

Also, was ist nötig, damit Sie etwas lernen? Erst einmal müssen Sie es *aufnehmen* und dann dafür sorgen, dass Sie es nicht wieder *vergessen*. Es geht nicht darum, Fakten in Ihren Kopf zu schieben. Nach den neuesten Forschungsergebnissen der Kognitionswissenschaft, der Neurobiologie und der Lernpsychologie gehört zum Lernen viel mehr als nur Text auf einer Seite. Wir wissen, was Ihr Gehirn anmacht.

Einige der Lernprinzipien dieser Buchreihe:



Wir setzen Bilder ein. An Bilder kann man sich viel besser erinnern als an Worte allein und lernt so viel effektiver (bis zu 89 % Verbesserung bei Abrufbarkeits- und Lerntransferstudien). Außerdem werden die Dinge dadurch verständlicher. **Wir setzen Text in oder neben die Grafiken,** auf die sie sich beziehen, anstatt darunter oder auf eine andere Seite. Die Leser werden auf den Bildinhalt bezogene Probleme dann mit *doppelt* so hoher Wahrscheinlichkeit lösen können.

Wir verwenden einen gesprächsorientierten Stil mit persönlicher Ansprache. Nach neueren Untersuchungen haben Studentinnen und Studenten nach dem Lernen bei Tests bis zu 40 % besser abgeschnitten, wenn der Inhalt den Leser direkt in der ersten Person und im lockeren Stil angesprochen hat statt in einem formalen Ton. Wir halten keinen Vortrag, sondern erzählen Ihnen Geschichten. Wir benutzen eine zwanglose Sprache. Wir versuchen, uns selbst nicht zu ernst zu nehmen. Denn: Würden *Sie* einer anregenden Unterhaltung beim Abendessen mehr Aufmerksamkeit schenken oder einem Vortrag?

Es ist wirklich ätzend, eine abstrakte Methode zu sein, so ganz ohne Rumpf ...

```
abstract void umherwandern();
```

Kein Methodenrumpf! Am Schluss steht ein Semikolon.

Wir wollen Sie dazu bringen, intensiver nachzudenken. Mit anderen Worten: Falls Sie nicht aktiv Ihre Neuronen strapazieren, passiert in Ihrem Gehirn nicht viel. Eine Leserin muss motiviert, begeistert und neugierig sein und angeregt werden, Probleme zu lösen, Schlüsse zu ziehen und sich neues Wissen anzueignen. Und dafür brauchen Sie Herausforderungen, Übungen, zum Nachdenken anregende Fragen und Tätigkeiten, die beide Seiten des Gehirns und mehrere Sinne einbeziehen.

Ergibt es einen Sinn zu sagen: Wanne IST-EIN Badezimmer? Badezimmer IST-EINE Wanne? Oder ist das eine HAT-EINE-Beziehung?

Wir versuchen, Ihre Aufmerksamkeit zu erlangen – und sie zu behalten. Wir alle haben schon Erfahrungen dieser Art gemacht: »Ich will das wirklich lernen, aber ich kann einfach nicht über Seite 1 hinaus wach bleiben.« Ihr Gehirn passt auf, wenn Dinge ungewöhnlich, interessant, merkwürdig, auffällig, unerwartet sind. Ein neues, schwieriges, technisches Thema zu lernen, muss nicht langweilig sein. Wenn es das nicht ist, lernt Ihr Gehirn viel schneller.

Wir sprechen Gefühle an. Wir wissen, dass Ihre Fähigkeit, sich an etwas zu erinnern, wesentlich von dessen emotionalem Gehalt abhängt. Sie erinnern sich an das, was Sie *bewegt*. Sie erinnern sich, wenn Sie etwas *fühlen*. Nein, wir erzählen keine herzzerreißenden Geschichten über einen Jungen und seinen Hund. Was wir erzählen, ruft Überraschungs-, Neugier-, Spaß- und Was-soll-das?-Emotionen hervor und dieses Hochgefühl, das Sie beim Lösen eines Puzzles empfinden oder wenn Sie etwas lernen, das alle anderen schwierig finden. Oder wenn Sie merken, dass Sie etwas können, das dieser »Ich bin ein besserer Techniker als du«-Typ aus der Technikabteilung *nicht kann*.

Metakognition: Nachdenken übers Denken

Wenn Sie wirklich lernen möchten, und zwar schneller und nachhaltiger, dann schenken Sie Ihrer Aufmerksamkeit Aufmerksamkeit. Denken Sie darüber nach, wie Sie denken. Lernen Sie, wie Sie lernen.

Die meisten von uns haben in ihrer Jugend keine Kurse in Metakognition oder Lerntheorie gehabt. Es wurde von uns *erwartet,* dass wir lernen, aber nur selten wurde uns auch *beigebracht,* wie man lernt.

Wir nehmen aber an, dass Sie wirklich Java lernen möchten, wenn Sie dieses Buch in den Händen halten. Und wahrscheinlich möchten Sie nicht viel Zeit aufwenden.

Wenn Sie so viel wie möglich von diesem Buch profitieren wollen oder von irgendeinem anderen Buch oder einer anderen Lernerfahrung, übernehmen Sie Verantwortung für Ihr Gehirn. Ihr Gehirn im Zusammenhang mit *diesem* Lernstoff.

Der Trick besteht darin, Ihr Gehirn dazu zu bringen, neuen Lernstoff als etwas wirklich Wichtiges anzusehen. Als entscheidend für Ihr Wohlbefinden. So wichtig wie ein Tiger. Andernfalls stecken Sie in einem dauernden Kampf, in dem Ihr Gehirn sein Bestes gibt, um die neuen Inhalte davon abzuhalten, hängen zu bleiben.

Wie bringen Sie also Ihr Gehirn dazu, Java so zu behandeln, als wäre es ein hungriger Tiger?

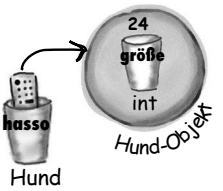
Da gibt es den langsamen, ermüdenden Weg oder den schnelleren, effektiveren Weg. Der langsame Weg führt über bloße Wiederholung. Natürlich ist Ihnen klar, dass Sie lernen und sich sogar an die langweiligsten Themen erinnern *können,* wenn Sie sich die gleiche Sache immer wieder einhämmern. Wenn Sie dasselbe nur oft genug wiederholen, sagt Ihr Gehirn: »Er hat zwar nicht das *Gefühl,* dass das wichtig ist, aber er sieht sich dieselbe Sache *immer und immer wieder* an – dann muss sie wohl wichtig sein.«

Der schnellere Weg besteht darin, ***alles zu tun, was die Gehirnaktivität erhöht***, vor allem verschiedene Arten von Gehirnaktivität. Eine wichtige Rolle dabei spielen die auf der vorhergehenden Seite erwähnten Dinge – alles Dinge, die nachweislich helfen, dass Ihr Gehirn *für* Sie arbeitet. So hat sich z. B. in Untersuchungen gezeigt: Wenn Wörter *in* den Abbildungen stehen, die sie beschreiben (und nicht irgendwo anders auf der Seite, z. B. in einer Bildunterschrift oder im Text), versucht Ihr Gehirn herauszufinden, wie die Wörter und das Bild zusammenhängen, und dadurch feuern mehr Neuronen. Und je mehr Neuronen feuern, umso größer ist die Chance, dass Ihr Gehirn mitbekommt: Bei dieser Sache lohnt es sich, aufzupassen, und vielleicht auch, sich daran zu erinnern.

Ein lockerer Sprachstil hilft, denn Menschen tendieren zu höherer Aufmerksamkeit, wenn ihnen bewusst ist, dass sie ein Gespräch führen – man erwartet dann ja von ihnen, dass sie dem Gespräch folgen und sich beteiligen. Das Erstaunliche daran ist: Es ist Ihrem Gehirn ziemlich egal, dass die »Unterhaltung« zwischen Ihnen und einem Buch stattfindet! Wenn der Schreibstil dagegen formal und trocken ist, hat Ihr Gehirn den gleichen Eindruck wie bei einem Vortrag, bei dem Sie in einem Raum voller passiver Zuhörer sitzen. Nicht nötig, wach zu bleiben.

Aber Abbildungen und ein lockerer Sprachstil sind erst der Anfang.

Das haben WIR getan:



Wir haben ***Bilder*** verwendet, weil Ihr Gehirn auf visuelle Eindrücke eingestellt ist, nicht auf Text. Soweit es Ihr Gehirn betrifft, sagt ein Bild *wirklich* mehr als 1.024 Worte. Und dort, wo Text und Abbildungen zusammenwirken, haben wir den Text *in* die Bilder eingebettet, denn Ihr Gehirn arbeitet besser, wenn der Text *innerhalb* der Sache steht, auf die er sich bezieht, und nicht in einer Bildunterschrift oder irgendwo vergraben im Text.

Wir haben ***Wiederholungen*** eingebaut, d. h. dasselbe auf *unterschiedliche* Art und mit verschiedenen Medientypen ausgedrückt, damit Sie es über *mehrere Sinne* aufnehmen. Das erhöht die Chance, dass die Inhalte an mehr als nur einer Stelle in Ihrem Gehirn verankert werden.

Wir haben Konzepte und Bilder in ***unerwarteter*** Weise eingesetzt, weil Ihr Gehirn auf Neuigkeiten programmiert ist. Und wir haben Bilder und Ideen mit zumindest *etwas* ***emotionalem*** *Charakter* verwendet, weil Ihr Gehirn darauf eingestellt ist, auf die Biochemie von Gefühlen zu achten. An alles, was ein *Gefühl* in Ihnen auslöst, können Sie sich mit höherer Wahrscheinlichkeit erinnern, selbst wenn dieses Gefühl nicht mehr ist als ein bisschen ***Belustigung, Überraschung*** oder ***Interesse.***

Wir haben einen ***umgangssprachlichen Stil*** mit direkter Anrede benutzt, denn Ihr Gehirn ist von Natur aus aufmerksamer, wenn es Sie in einer Unterhaltung wähnt als wenn es davon ausgeht, dass Sie passiv einer Präsentation zuhören – sogar dann, wenn Sie *lesen.*

Wir haben mehr als 50 ***Übungen*** für Sie vorgesehen, denn Ihr Gehirn lernt und behält von Natur aus besser, wenn Sie Dinge ***tun,*** als wenn Sie nur darüber *lesen.* Und wir haben die Übungen zwar anspruchsvoll, aber doch lösbar gemacht, denn so ist es den meisten *Lesern* am liebsten.

Wir haben ***mehrere unterschiedliche Lernstile*** verwendet, denn vielleicht bevorzugen *Sie* ein Schritt-für-Schritt-Vorgehen, während ein anderer erst einmal den groben Zusammenhang verstehen und eine Dritte einfach nur ein Codebeispiel sehen möchte. Aber ganz abgesehen von den jeweiligen Lernvorlieben profitieren *alle* davon, wenn sie die gleichen Inhalte in unterschiedlicher Form präsentiert bekommen.

Punkt für Punkt

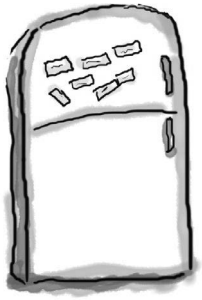
Wir liefern Inhalte für ***beide Seiten Ihres Gehirns,*** denn je mehr von Ihrem Gehirn Sie einsetzen, umso wahrscheinlicher werden Sie lernen und behalten und umso länger bleiben Sie konzentriert. Wenn Sie mit einer Seite des Gehirns arbeiten, bedeutet das häufig, dass sich die andere Seite des Gehirns ausruhen kann; so können Sie über einen längeren Zeitraum produktiver lernen.

Und wir haben ***Geschichten*** und Übungen aufgenommen, die ***mehr als einen Blickwinkel repräsentieren,*** denn Ihr Gehirn lernt von Natur aus intensiver, wenn es gezwungen ist, selbst zu analysieren und zu beurteilen.

Wir haben ***Herausforderungen*** eingefügt: in Form von Übungen und indem wir ***Fragen*** stellen, auf die es nicht immer eine eindeutige Antwort gibt, denn Ihr Gehirn ist darauf eingestellt, zu lernen und sich zu erinnern, wenn es an etwas *arbeiten* muss (so wie Sie ja auch Ihren *Körper* nicht in Form bekommen können, indem Sie andere auf dem Sportplatz *beobachten).* Aber wir haben unser Bestes getan, um dafür zu sorgen, dass Sie – wenn Sie schon hart arbeiten – an den *richtigen* Dingen arbeiten. Dass Sie ***nicht einen einzigen Dendriten darauf verschwenden,*** ein schwer verständliches Beispiel zu verarbeiten oder einen schwierigen, mit Fachbegriffen gespickten oder extrem gedrängten Text zu analysieren.

Wir haben ***Menschen*** eingesetzt. In Geschichten, Beispielen, Bildern usw. – denn *Sie sind* ein Mensch. Und Ihr Gehirn schenkt *Menschen* mehr Aufmerksamkeit als *Dingen.*

Und wir haben einen ***80/20***-Ansatz genutzt. Wir gehen davon aus, dass dies nicht Ihr einziges Buch sein wird, wenn Sie einen Doktor in Java machen wollen. Deshalb besprechen wir nicht *alles.* Nur das, was Sie wirklich *verwenden* werden.



Schneiden Sie dies aus und heften Sie es an Ihren Kühlschrank.

Und das können SIE tun, um sich Ihr Gehirn untertan zu machen

So, wir haben unseren Teil der Arbeit geleistet. Der Rest liegt bei Ihnen. Diese Tipps sind ein Anfang; hören Sie auf Ihr Gehirn und finden Sie heraus, was bei Ihnen funktioniert und was nicht. Probieren Sie neue Wege aus.

1 Immer langsam. Je mehr Sie verstehen, umso weniger müssen Sie auswendig lernen.

Lesen Sie nicht nur. Halten Sie inne und denken Sie nach. Wenn das Buch Sie etwas fragt, springen Sie nicht einfach zur Antwort. Stellen Sie sich vor, dass Sie das wirklich jemand *fragt.* Je gründlicher Sie Ihr Gehirn zum Nachdenken zwingen, umso größer ist die Chance, dass Sie lernen und behalten.

2 Bearbeiten Sie die Übungen. Machen Sie selbst Notizen.

Wir haben sie entworfen, aber wenn wir sie auch für Sie lösen, wäre das so, als würde jemand anderes Ihr Training für Sie absolvieren. Und sehen Sie sich die Übungen *nicht einfach nur an.* **Benutzen Sie einen Bleistift.** Es deutet vieles darauf hin, dass körperliche Aktivität *beim* Lernen den Lernerfolg erhöhen kann.

3 Lesen Sie die Abschnitte »Es gibt keine Dummen Fragen«.

Und zwar alle. Das sind keine Zusatzanmerkungen – sie gehören zum Kerninhalt! Aus den Fragen lernen Sie manchmal noch mehr als aus den Antworten.

4 Lesen Sie nicht alles am selben Ort.

Stehen Sie auf, recken Sie sich, gehen Sie herum, wechseln Sie den Stuhl, gehen Sie in einen anderen Raum. Das trägt dazu bei, dass Ihr Gehirn etwas *fühlt,* und verhindert, dass Ihr Lernerfolg an einen bestimmten Ort gebunden ist.

5 Lesen Sie dies als Letztes vor dem Schlafengehen. Oder lesen Sie danach zumindest nichts *Anspruchsvolles* mehr.

Ein Teil des Lernprozesses (vor allem die Übertragung in das Langzeitgedächtnis) findet erst statt, *nachdem* Sie das Buch zur Seite gelegt haben. Ihr Gehirn braucht Zeit für sich, um weitere Verarbeitung zu leisten. Wenn Sie in dieser Zeit etwas Neues aufnehmen, geht ein Teil dessen, was Sie gerade gelernt haben, verloren.

6 Trinken Sie Wasser. Viel.

Ihr Gehirn arbeitet am besten in einem schönen Flüssigkeitsbad. Austrocknung (zu der es schon kommen kann, bevor Sie überhaupt Durst verspüren) beeinträchtigt die kognitive Funktion.

7 Reden Sie drüber. Laut.

Sprechen aktiviert einen anderen Teil des Gehirns. Wenn Sie etwas verstehen möchten oder Ihre Chancen verbessern wollen, sich später daran zu erinnern, sagen Sie es laut. Noch besser: Versuchen Sie, es jemand anderem laut zu erklären. Sie lernen dann schneller und haben vielleicht Ideen, auf die Sie beim bloßen Lesen nie gekommen wären.

8 Hören Sie auf Ihr Gehirn.

Achten Sie darauf, Ihr Gehirn nicht zu überladen. Wenn Sie merken, dass Sie etwas nur noch überfliegen oder dass Sie das gerade erst Gelesene vergessen haben, ist es Zeit für eine Pause. Ab einem bestimmten Punkt lernen Sie nicht mehr schneller, indem Sie mehr hineinzustopfen versuchen; das kann sogar den Lernprozess stören.

9 Aber bitte mit *Gefühl*!

Ihr Gehirn muss wissen, dass es *um etwas Wichtiges geht.* Lassen Sie sich in die Geschichten hineinziehen. Erfinden Sie eigene Bildunterschriften für die Fotos. Über einen schlechten Scherz zu stöhnen, ist *immer noch* besser, als gar nichts zu fühlen.

10 Tippen Sie den Code ab und führen Sie ihn aus.

Tippen Sie die Codebeispiele ab und führen Sie sie aus. Dann können Sie herumexperimentieren und den Code verändern und verbessern (oder ihn kaputtmachen – das ist manchmal der beste Weg, um herauszufinden, was wirklich passiert). Bei langen Beispielen oder dem »Fertiggericht«-Code können Sie die Quellcodedateien von der Webseite zum Buch aus dem Onlinekatalog von O'Reilly herunterladen.

Was Sie für dieses Buch brauchen:

Was Sie *nicht* brauchen, ist ein zusätzliches Entwicklungswerkzeug wie z. B. eine integrierte Entwicklungsumgebung (Integrated Development Environment, kurz: IDE). Wir empfehlen Ihnen nachdrücklich, *ausschließlich* einen einfachen Texteditor zu verwenden, solange Sie dieses Buch noch nicht durchhaben. Eine IDE kann Ihnen einige der Details vorenthalten, die aber wirklich wichtig sind. Daher ist es viel besser, wenn Sie das Ganze zunächst auf der Kommandozeile lernen. Erst wenn Sie die Vorgänge wirklich verstanden haben, sollten Sie auf ein Tool umsteigen, das einen Teil des Prozesses automatisiert.

In diesem Buch gehen wir davon aus, dass Sie mindestens Java 11 einsetzen (mit Ausnahme von Anhang B). Sollten Sie mit Java 8 arbeiten, wird der größte Teil Ihres Codes aber trotzdem funktionieren.

Wenn wir ein Merkmal einer höheren Version als Java 8 besprechen, weisen wir auf die benötigte Version hin.

JAVA INSTALLIEREN

- Da sich die Versionen und damit auch die Empfehlung für das richtige JDK schnell ändern, haben wir die detaillierten Anweisungen zur Installation von Java als Teil der Codebeispiele online zur Verfügung gestellt:

  https://oreil.ly/hfJava_install

  Dies ist eine vereinfachte Version davon.
- Wenn Sie nicht wissen, welche Java-Version Sie verwenden sollen, empfehlen wir Java 17.
- Es gibt viele kostenlose Builds des OpenJDK (der Open-Source-Version von Java). Wir empfehlen das von der Gemeinschaft getragene Eclipse Adoptium JDK, zu finden unter *https://adoptium.net*.
- Das JDK enthält alles, was Sie zum Kompilieren und Ausführen von Java brauchen. Es enthält allerdings keine API-Dokumentation. Die brauchen Sie aber! Laden Sie die Java-SE-API-Dokumentation herunter. Alternativ können Sie auch online auf die Dokumentation zugreifen, aber glauben Sie uns – der Download lohnt sich!
- Sie benötigen einen Texteditor. So gut wie jeder Editor ist geeignet (vi, emacs), auch die GUI-basierten Programme, die den meisten Betriebssystemen beiliegen. Notepad, Wordpad, TextEdit etc. funktionieren alle, solange Sie im Nur-Text-Modus (**nicht** Rich Text) arbeiten und sicherstellen, dass nicht ».txt« am Ende Ihres Quellcodedateinamens (».java«) angehängt wird.
- Nach dem Herunterladen/Entpacken/wasauchimmer (das hängt von der Version und Ihrem Betriebssystem ab) müssen Sie einen Eintrag für Ihre PATH-Umgebungsvariable vornehmen, die auf das *bin*-Verzeichnis innerhalb des Java-Hauptverzeichnisses verweist. Das *bin*-Verzeichnis ist das Verzeichnis, zu dem Sie einen PFAD (engl. Path) brauchen. Wenn Sie auf der Kommandozeile

  ```
  % javac
  ```

  eingeben, weiß Ihr Terminal auf diese Weise, wo es den javac-Compiler findet.
- Hinweis: Sofern Sie Schwierigkeiten bei der Installation haben, empfehlen wir Ihnen, das Forum für Java-Anfänger auf javaranch.com aufzusuchen! Aber eigentlich sollten Sie das auf jeden Fall tun – egal ob Sie Probleme haben oder nicht.

Den Original-Beispielcode für dieses Buch finden Sie unter: *https://oreil.ly/hfJava_3e_examples*.

Was Sie sonst noch wissen sollten:

Dies ist ein Lernerlebnis, kein Nachschlagewerk. Wir haben bewusst alles herausgestrichen, was an irgendeiner Stelle des Buchs hinderlich für den *Lernprozess* sein könnte. Und wenn Sie das Buch das erste Mal durcharbeiten, müssen Sie am Anfang beginnen, denn das Buch geht stets davon aus, dass Sie bestimmte Dinge bereits gesehen und gelernt haben.

Wir verwenden einfache, UML-ähnliche Diagramme.

Wenn wir *reines* UML verwendet hätten, würden Sie zwar etwas sehen, das wie Java *aussieht*, aber mit einer Syntax, die schlicht und einfach *falsch* ist. Deshalb haben wir eine vereinfachte UML-Version genutzt, die nicht im Widerspruch zur Java-Syntax steht. Falls Sie UML noch nicht kennen, müssen Sie sich hier nicht damit abplagen, Java *und* UML gleichzeitig zu lernen.

Wir verwenden ein einfacheres, modifiziertes Pseudo-UML.

Wir kümmern uns nicht um das Organisieren und Verpacken Ihres Codes.

In diesem Buch können Sie sich in aller Ruhe dem Lernen von Java widmen, ohne mit bestimmten Organisations- oder Verwaltungsdetails der Entwicklung von Java-Programmen belastet zu werden. Später, in der »echten« Welt, müssen Sie diese Details kennen und damit arbeiten, aber da das Bereitstellen von Java-Anwendungen im Allgemeinen auf Build-Tools von Drittanbietern wie Maven und Gradle angewiesen ist, sind wir davon ausgegangen, dass Sie diese Tools separat erlernen werden.

Die Spitzen-Sie-Ihren-Bleistift-Aufgaben sollten Sie ALLE zu lösen versuchen.

Die Übungen am Kapitelende sind notwendig; die Rätsel sind optional. Die jeweiligen Antworten finden Sie ganz zum Schluss des Kapitels.

Eins müssen Sie über die »Puzzles« und »Rätsel« wissen – *es sind wirklich Rätsel*. So wie Knobeleien, Denksportaufgaben, Kreuzworträtsel usw. Die *Übungen* sind dazu gedacht, das Gelernte einzuüben, und sollten daher auf keinen Fall ausgelassen werden. Das gilt nicht für die Rätsel; manche davon stellen sogar eine ziemliche Herausforderung dar. Sie sind etwas für *Rätselfreunde* – und ob Sie ein Rätselfreund sind oder nicht, wissen Sie sicher selbst. Wenn Sie es nicht genau wissen, schlagen wir Ihnen vor, ein paar davon auszuprobieren, aber lassen Sie sich auf keinen Fall entmutigen, wenn Sie ein Rätsel oder Puzzle *nicht* lösen können oder einfach nicht genug Zeit dafür aufwenden wollen.

Aufgaben, die mit dem Trainingslogo (den Laufschuhen) gekennzeichnet sind, sind Pflicht! Überspringen Sie sie nicht, wenn Sie es ernst meinen mit dem Erlernen von Java.

Zu den »Spitzen Sie Ihren Bleistift«-Übungen gibt es keine Lösungen.

Zumindest sind diese nicht im Buch abgedruckt. Für manche gibt es keine eindeutige Antwort, und bei anderen sollten *Sie* im weiteren Verlauf Ihrer Lernerfahrung selbst entscheiden, ob *Ihre* Antworten richtig waren.

Die Codebeispiele sind so kurz wie möglich.

Es ist frustrierend, sich durch 200 Zeilen Code zu wühlen, um die beiden Zeilen zu finden, die man zum Verständnis benötigt. Bei den meisten Beispielen in diesem Buch wird vom Kontext so wenig abgedruckt wie möglich – so wird der Teil, den Sie gerade lernen wollen, klar und einfach. Erwarten Sie vom Code nicht unbedingt, dass er stabil läuft oder vollständig ist. Das bleibt als Aufgabe für *Sie*, sobald Sie mit dem Buch durch sind. Die Beispiele im Buch wurden speziell zum Zweck des *Lernens* geschrieben und sind nicht immer voll funktionsfähig.

Wenn Sie das Puzzle-Logo sehen, geht es um etwas Freiwilliges, und wenn Sie keinen Spaß an Knobeleien oder Kreuzworträtseln haben, werden Ihnen auch diese nicht gefallen.

Die Fachgutachter der dritten englischen Auflage

Marc Loy

Marc begann seine Java-Ausbildung in der Frühzeit bei Sun Microsystems (ein Lob an HotJava!) und ist seitdem erfolgreich dabei. Er ist Autor einer Reihe früher Java-Bücher und Trainingskurse und arbeitete dabei mit einer Vielzahl unterschiedlicher Unternehmen aus den USA, Europa und Asien zusammen. Zuletzt schrieb Marc für O'Reilly das Buch *Smaller C* und war Co-Autor der fünften englischen Auflage von *Learning Java*. Aktuell arbeitet Marc in Ohio als Entwickler und Maker, der sich auf Mikrocontroller spezialisiert hat.

Abraham Marin-Perez

Abraham ist Java-Programmierer, Berater, Autor und Vortragsredner mit über zehn Jahren Erfahrung in einer Vielzahl von Branchen. Ursprünglich aus Valencia in Spanien stammend, hat Abraham den Großteil seiner Karriere in London verbracht, wo er für Kunden wie JP Morgan oder das britische Innenministerium arbeitet, häufig in Zusammenarbeit mit Equal Experts. Aus der Überzeugung, dass seine Erfahrungen für andere nützlich sein könnten, engagierte sich Abraham als Java-Nachrichtenredakteur bei InfoQ. Er ist außerdem Autor von *Real-World Maintainable Software* und Co-Autor von *Continuous Delivery in Java*. Daneben hilft er bei der Leitung der Londoner Java-Community. Als ewig Lernender strebt Abraham außerdem einen Abschluss in Physik an.

Weitere Menschen, denen wir für die dritte englische Auflage Dank schulden

Bei O'Reilly:

Ein riesiges Dankeschön an **Zan McQuade** und **Nicole Taché** dafür, dass sie uns die Möglichkeit gaben, diese Auflage endlich herauszubringen! Zan, danke, dass du Trisha wieder mit der Head-First-Welt zusammengebracht hast, und Nicole: fantastische Arbeit, uns dazu zu bringen, dieses Buch zu Ende zu bringen. Danke auch an **Meghan Blanchette**, die O'Reilly schon vor 100 Jahren verlassen hat, aber die Bert und Trisha anno 2014 miteinander bekannt gemacht hat.

Trisha möchte sich gerne bedanken bei:

Helen Scott für ihre häufigen Rückmeldungen zu den neuen Themen im Buch. Sie hat mich immer wieder davon abgehalten, zu tief in die Materie einzutauchen oder zu viel Wissen vorauszusetzen. Sie ist eine echte Fürsprecherin der Lernenden. Ich kann es kaum erwarten, bei unserem nächsten Projekt noch enger mit ihr zusammenzuarbeiten.

Meinem Team bei JetBrains für ihre Geduld und ihre Ermutigungen: **Dalia Abo Sheasha** für das Testen des Kapitels über Lambdas und Streams und **Mala Gupta** dafür, dass sie mir genau die Informationen gab, die ich über moderne Java-Zertifizierungen brauchte. Ein besonders großes Dankeschön an **Hadi Hariri** für seine ununterbrochene Unterstützung.

Den Friday Pub Lunch *informaticos* dafür, dass sie alle mittäglichen Gespräche zu jedem Java-Thema toleriert haben, das ich an dem Wochentag gerade zu erklären versuchte. Danke auch an **Alys**, **Jen** und **Clare** für ihre Hilfe, als es darum ging, herauszufinden, wann dieses Buch Vorrang vor der Familie hat. Danke an **Holly Cummins**, die buchstäblich im letzten Moment noch einen Bug fand.

Evie und **Amy** für ihre Verbesserungsvorschläge für die Eiscreme-Beispiele zu Javas Optional-Typen. Danke euch beiden für euer ehrliches Interesse an meinen Fortschritten und die spontanen High-fives, als ihr mitbekommen habt, dass ich endlich fertig war.

Nichts hiervon wäre ohne **Israel Boza Rodriguez** möglich gewesen. Du hast es ausgehalten, dass ich von wichtigen Gesprächen wie: »Was sollen wir zu Abend essen?« mit Fragen wie: »Meinst du, CountDownLatch ist zu nischenhaft, um es beginnenden Entwicklern beizubringen?« ablenkte. Vor allem hast du mir geholfen, Platz und Zeit für die Arbeit an diesem Buch zu schaffen, und mich regelmäßig daran erinnert, warum ich dieses Projekt überhaupt angenommen habe.

Herzlichen Dank an **Bert** und **Kathy** dafür, dass sie mich auf diese Reise mitnahmen. Es war mir eine Ehre, zu erfahren, wie man sozusagen aus erster Hand lernt, eine Head-First-Autorin zu sein.

Bert und Kathy möchten sich bedanken bei:

Beth Robson und **Eric Freeman** für ihre umfassende und andauernde großartige Unterstützung der Head-First-Buchreihe. Ein besonderes Dankeschön an Beth für die vielen Gespräche, in denen wir diskutierten, welche neuen Java-Themen wir auf welche Weise vermitteln sollten.

Paul Wheaton und den erstaunlichen Moderatoren von CodeRanch.com (alias JavaRanch), die dafür sorgen, dass CodeRanch ein freundlicher Ort für Java-Anfänger bleibt. Besonderer Dank gilt **Campbell Ritchie**, **Jeanne Boyarsky**, **Stephan van Hulst**, **Rob Spoor**, **Tim Cooke**, **Fred Rosenberger und Frits Walraven** für ihren unschätzbaren Rat dazu, was seit der zweiten englischen Auflage die wirklich wichtigen Erweiterungen von Java waren.

Dave Gustafson dafür, dass er mir so viel über Softwareentwicklung und Rock Climbing beigebracht hat, UND für großartige Diskussionen über den aktuellen Stand der Programmierung. **Eric Normand** dafür, dass er uns allen ein bisschen funktionale Programmierung beigebracht und so geholfen hat, einige der besten Ideen der FP in einem OO-Buch unterzubringen. **Simon Roberts** dafür, dass er Studierenden auf der ganzen Welt mit Ausdauer und Leidenschaft Java beibringt. Danke an **Heinz Kabutz** und **Venkat Subramaniam** für ihre Hilfe bei der Erforschung der Haken und Ösen von Java Streams.

Laura Baldwin und **Mike Loukides** für ihre unermüdliche Unterstützung von Head First während all der Jahre.
Ron Bilodeau und **Kristen Brown** für ihre hervorragende, immer geduldige und freundliche Unterstützung.

Die Fachgutachter der zweiten englischen Auflage

Unendlichen Dank an Jessica und Val für ihre harte Arbeit an der zweiten englischen Auflage.

Jess arbeitet bei Hewlett-Packard im Self Healing Services Team. Sie machte ihren Bachelor-Abschluss in Technischer Informatik an der Villanova University, hat SCJP-1.4- und SCWCD-Zertifizierungen und steht kurz vor ihrem Master in Technischer Informatik an der Drexel University (puh!).

Wenn Jess nicht arbeitet, studiert oder in ihrem MINI Cooper S unterwegs ist, kämpft sie vielleicht gerade mit ihrer Katze um das Garn, mit dem sie ihr neuestes Strick- oder Häkelprojekt fertigstellen möchte (will jemand einen Hut?). Ursprünglich stammt sie aus Salt Lake City, Utah (nein, sie ist keine Mormonin ... ja, auch Sie wollten das fragen ...) und lebt zurzeit mit ihrem Mann Mendra und zwei Katzen, Chai und Sake, in der Nähe von Philadelphia.

Begegnen können Sie ihr beim Moderieren der technischen Foren auf *javaranch.com*.

Valentin machte seinen Master-Abschluss in Informatik an der Schweizer École Polytechnique Fédérale de Lausanne (EPFL). Er hat als Entwickler bei SRI International (Menlo Park, Kalifornien) und als leitender Entwickler im Software Engineering Laboratory der EPFL gearbeitet.

Valentin ist Mitbegründer und technischer Direktor von Condris Technologies, einer Firma, die sich auf die Entwicklung von Softwarearchitekturlösungen spezialisiert hat.

Zu seinen Forschungs- und Entwicklungsinteressen gehören aspektorientierte Programmierung, Ent- wurfs- und Architekturmuster, Webdienste und Softwarearchitektur. Valentin kümmert sich um seine Frau, um die Gartenarbeit, liest, treibt Sport – und neben all dem moderiert er auch noch die SCBCD- und SCDJWS-Foren auf *javaranch.com*. Er hat SCJP-, SCJD-, SCBCD-, SCWCD- und SCDJWS- Zertifizierungen. Zudem hatte er auch die Gelegenheit, als Co-Autor für den Whizlabs SCBCD Exam Simulator zu fungieren.

(Wir stehen immer noch unter Schock, dass wir ihn mit einer *Krawatte* sehen mussten.)

Dank für die zweite englische Auflage schulden wir

~~Schuld haben~~ außerdem:

Bei O'Reilly:

Unser besonderer Dank geht an **Mike Loukides** bei O'Reilly, der das Risiko dieses Buchs eingegangen ist und uns geholfen hat, aus dem »Head First«-Konzept ein Buch (beziehungsweise eine Buch*reihe*) zu entwickeln. Wenn diese zweite Auflage in den Druck geht, gibt es bereits fünf Bücher aus der Reihe »Head First«, und Mike hat uns auf dem ganzen Weg begleitet. Wir danken auch **Tim O'Reilly** für seine Bereitschaft, sich auf etwas *völlig* Neues und Andersartiges einzulassen. Und der klugen **Kyle Hart** dafür, dass sie herausgefunden hat, wie »Head First« in diese Welt passt, und für die Markteinführung der Reihe. Und schließlich **Edie Friedman** für das »Kopfbetonende« Cover der Reihe.

Unseren tapferen Betatestern und Review-Team-Mitgliedern:

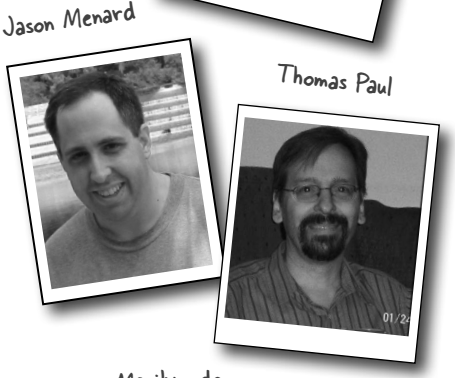
Ganz besonders danken wir dem Leiter unseres JavaRanch-Review-Teams, **Johannes de Jong**. Du bist jetzt zum fünften Mal bei einem unserer »Head First«-Bücher dabei, und wir freuen uns mächtig, dass du trotzdem noch mit uns sprichst. **Jeff Cumps** ist zum dritten Mal dabei und findet unerbittlich die Stellen, an denen wir deutlicher oder exakter sein müssen.

Corey McGlone, du bist spitze. Und wir finden, du gibst auf javaranch.com die besten Erklärungen. Wahrscheinlich hast du gemerkt, dass wir eine oder zwei davon geklaut haben. **Jason Menard** hat uns mehr als einmal – und nicht nur bei Kleinigkeiten – in fachlicher Hinsicht gerettet, und **Thomas Paul** hat uns wie immer fachliches Feedback geliefert und verzwickte Java-Probleme gefunden, die wir anderen übersehen hatten. **Jane Griscti**, Java-Programmiererin mit Brief und Siegel (die zudem das eine oder andere übers Schreiben weiß), war uns – gemeinsam mit dem altgedienten JavaRancher **Barry Gaunt** – eine große Hilfe bei der Neuauflage.

Marilyn de Queiroz hat uns bei beiden Auflagen des Buchs großartig unterstützt. Bei der ersten Auflage waren uns außerdem **Chris Jones**, **John Nyquist**, **James Cubeta**, **Terri Cubeta** und **Ira Becker** eine riesige Hilfe.

Besonderer Dank geht an einige der Head-First-Leute, die uns von Anfang an hilfreich zur Seite gestanden haben: **Angelo Celeste**, **Mikalai Zaikin** und **Thomas Duff** (*twduff.com*). Und danke an unseren wunderbaren Agenten, **David Rogelberg** von StudioB (aber mal ernsthaft, was ist mit den *Filmrechten*?).

Ein paar der Java-Expertinnen und -Experten aus unserem Review-Team ...



Gerade als Sie dachten, es kämen keine weiteren Danksagungen* ...

Noch mehr Java-Expertinnen und -Experten, die bei der ersten Auflage geholfen haben (in pseudozufälliger Reihenfolge):

Emiko Hori, Michael Taupitz, Mike Gallihugh, Manish Hatwalne, James Chegwidden, Shweta Mathur, Mohamed Mazahim, John Paverd, Joseph Bih, Skulrat Patanavanich, Sunil Palicha, Suddhasatwa Ghosh, Ramki Srinivasan, Alfred Raouf, Angelo Celeste, Mikalai Zaikin, John Zoetebier, Jim Pleger, Barry Gaunt und Mark Dielen.

Das Puzzle-Team der ersten Auflage:

Dirk Schreckmann, Mary »Java-Kreuzworträtsel-Champion« Leners, Rodney J. Woodruff, Gavin Bong und Jason Menard. JavaRanch hat wirklich Glück, dass ihr alle dort mithelft.

Weitere Mitverschwörer, denen wir danken möchten, sind:

Paul Wheaton, der Ober-Cowboy von javaranch.com, der Tausende von Java-Anfängern unterstützt hat.

Solveig Haugland, J2EE-Meisterin und Autorin von »Dating Design Patterns«.

Die Autoren **Dori Smith** und **Tom Negrino** (**backupbrain.com**), die uns durch die Welt der technischen Fachbücher gelotst haben.

Unsere Head-First-Komplizen, **Eric Freeman und Beth Freeman** (die Autoren von »Head First Design Patterns« – in deutscher Übersetzung: »Entwurfsmuster von Kopf bis Fuß«), die uns Bawls™ zu trinken gaben und uns dadurch den nötigen Koffeinschub verpassten, um das hier pünktlich fertigzustellen.

Sherry Dorris für die Dinge, auf die es wirklich ankommt.

Die mutigen »Early Adopters« dieser Reihe:

Joe Litton, Ross P. Goldberg, Dominic Da Silva, honestpuck, Danny Bromberg, Stephen Lepp, Elton Hughes, Eric Christensen, Vulinh Nguyen, Mark Rau, Abdulhaf, Nathan Oliphant, Michael Bradly, Alex Darrow, Michael Fischer, Sarah Nottingham, Tim Allen, Bob Thomas und Mike Bibby (der Erste).

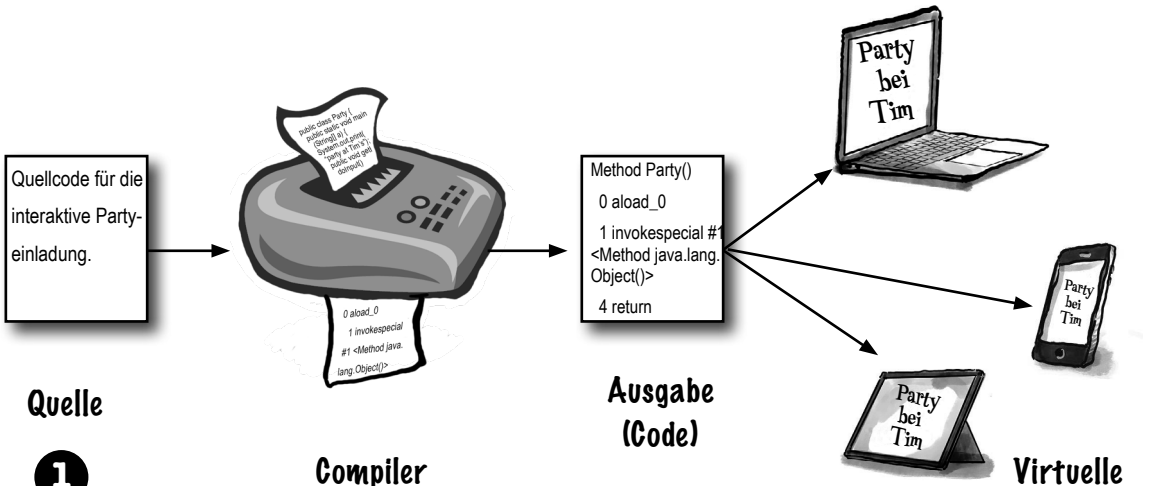
*Die große Zahl der Danksagungen hat folgenden Grund: Wir testen die Theorie, dass jeder, der in einem Buch bei den Danksagungen erwähnt wird, mindestens ein Exemplar davon kauft, wahrscheinlich sogar mehrere, wegen der Verwandtschaft und so. Wenn Sie bei den Danksagungen in unserem nächsten Buch erwähnt werden möchten und eine große Familie haben, schreiben Sie uns!

Die Oberfläche durchbrechen

Java bringt Sie an neue Orte. Seit seiner bescheidenen Veröffentlichung in der (schwächlichen) Version 1.02 hat Java Programmierer mit seiner freundlichen Syntax, seinen objektorientierten Features, der Speicherverwaltung und vor allem mit dem Versprechen auf Portabilität verführt. Die Verlockung des **Write-once, Run-anywhere** ist einfach zu groß. Es entstand eine treue Anhängerschaft, die explosionsartig wuchs, während Programmierer gegen Bugs, Einschränkungen und vor allem dagegen kämpften, dass es schneckenlangsam war. Aber das ist schon lange her. Wenn Sie gerade erst mit Java beginnen, **haben Sie Glück**. Einige von uns mussten sich zehn Kilometer durch den Schnee kämpfen (barfuß), in jede Richtung bergauf, um selbst die trivialste Applikation ans Laufen zu bringen. *Sie* dagegen können sich von dem **geschmeidigeren, schnelleren, viel, viel mächtigeren** Java von heute tragen lassen.

Wie Java funktioniert

Unser Ziel ist es, eine Applikation zu schreiben (in unserem Fall eine interaktive Partyeinladung), die auf jedem beliebigen von Ihren Freunden verwendeten Gerät funktionieren soll.



Quelle

1

Erstellen Sie ein Quelldokument. Verwenden Sie ein etabliertes Protokoll (in diesem Fall die Sprache Java).

Compiler

2

Lassen Sie Ihr Dokument durch einen Quellcode-Compiler laufen. Er überprüft den Code auf Fehler und lässt Sie erst kompilieren, wenn er sicher ist, dass alles richtig läuft.

Ausgabe (Code)

3

Der Compiler erstellt ein neues Dokument, das als Java-***Bytecode*** codiert ist. Jedes Gerät, das Java ausführen kann, kann auch diese Datei interpretieren bzw. in etwas übersetzen, das es ausführen kann. Der kompilierte Bytecode ist plattformunabhängig.

Virtuelle Maschinen

4

Auf den elektronischen Spielzeugen Ihrer Freunde läuft eine als Software implementierte ***virtuelle*** Java-Maschine (JVM). Wenn Ihre Freunde das Programm laufen lassen, liest die virtuelle Maschine den Bytecode und führt ihn aus.

Was Sie in Java tun werden

Sie schreiben eine Quellcode-Datei, kompilieren sie mit dem javac-Compiler und führen dann den kompilierten Bytecode in einer virtuellen Java-Maschine aus.

```
import java.awt.*;
import java.awt.event.*;

class Party {
 public void buildInvite() {
  Frame f = new Frame();
  Label l = new Label("Party bei Tim");
  Button b = new Button("Sicher doch");
  Button c = new Button("Ohne mich");
  Panel p = new Panel();
  p.add(l);
 }  // mehr Code ….
}
```

Quelle

1

Geben Sie Ihren Quellcode ein.

Speichern Sie die Datei als ***Party.java***.

Compiler

2

Kompilieren Sie die Datei ***Party.java***, indem Sie `javac` (die Compiler-Anwendung) ausführen. Wenn Sie keine Fehler gemacht haben, erhalten Sie ein zweites Dokument namens ***Party.class***.

Die vom Compiler erzeugte Datei ***Party.class*** enthält *Bytecode*.

```
Method Party()
  0 aload_0
  1 invokespecial #1 <Method
java.lang.Object()>
  4 return
Method void buildInvite()
  0 new #2 <Class java.awt.Frame>
  3 dup
  4 invokespecial #3 <Method
java.awt.Frame()>
```

Ausgabe (Code)

3

Kompilierter Code: ***Party.class***

Virtuelle Maschinen

4

Führen Sie das Programm aus, indem Sie die Java Virtual Machine (JVM) mit der Datei ***Party.class*** starten. Die JVM übersetzt den *Bytecode* in etwas, das die zugrunde liegende Plattform versteht, und führt Ihr Programm aus.

(Hinweis: Dies soll **KEINE** Anleitung sein … den echten Code werden Sie gleich schreiben. Jetzt sollen Sie erst mal ein Gefühl dafür bekommen, wie alles zusammenpasst.

Anders gesagt, der Code auf dieser Seite ist kein echter Code. Versuchen Sie nicht, ihn auszuführen.)

Eine sehr kurze Geschichte von Java

Java wurde erstmals am 26. Januar 1996 veröffentlicht (manche Menschen meinen eher, es sei »ausgebrochen«). Die Sprache ist also schon über 25 Jahre alt! Während der ersten 25 Jahre hat sich Java als Sprache weiterentwickelt, und die JAVA-API ist enorm gewachsen. Unserer besten Schätzung nach wurden in den vergangenen Jahren ungefähr 17 Trillionen Zeilen Code geschrieben. Beim Programmieren wird Ihnen daher mit großer Sicherheit Code begegnen, der schon ziemlich alt ist, anderer wird deutlich neuer sein. Java ist berühmt für seine Rückwärtskompatibilität, sodass alter Code auch noch auf neuen JVMs ausgeführt werden kann.

In diesem Buch beginnen wir meist mit einem eher älteren Programmierstil (Code dieser Art wird Ihnen sehr wahrscheinlich auch in der »wahren Welt« begegnen), und danach folgt Code im neuen Stil.

Entsprechend zeigen wir Ihnen manchmal ältere Klassen der Java-API und gehen dann auf neuere Alternativen ein.

Ich habe gehört, dass Java im Vergleich mit anderen kompilierten Sprachen wie C und Rust nicht besonders schnell ist.

Geschwindigkeit und Speicherverbrauch

Bei seiner ersten Veröffentlichung war Java noch ziemlich langsam. Kurz danach wurde jedoch die HotSpot-VM entwickelt, und es wurden andere Maßnahmen zu Leistungssteigerung getroffen. Auch wenn Java nicht schnellste aller Sprachen ist, gilt sie als sehr schnell – fast so schnell wie C und Rust und ***viel*** schneller als die meisten anderen Sprachen da draußen.

Java besitzt eine magische Superkraft – die JVM. Die Java Virtual Machine kann Ihren Code optimieren, ***während er ausgeführt wird***. Dadurch kann man sehr schnelle Anwendungen erstellen, ohne hierfür erst speziellen Hochleistungscode zu schreiben.

Aber – um die Hosen komplett herunterzulassen – im Vergleich zu C und Rust verbraucht Java eine Menge Arbeitsspeicher.

Spitzen Sie Ihren Bleistift

→ Antworten auf Seite 6.

Sehen Sie, wie einfach es ist, Java zu schreiben

Versuchen Sie zu erraten, was die einzelnen Codezeilen tun (die Antworten finden Sie auf der folgenden Seite).

`int size = 27;`	Deklariert eine Integer-Variable namens »size« und weist ihr den Wert 27 zu.
`String name = "Fido";`	
`Dog myDog = new Dog(name, size);`	
`x = size - 5;`	
`if (x < 15) myDog.bark(8);`	Ist x (mit dem Wert 22) kleiner als 15, wird der Hund angewiesen, achtmal zu bellen.
`while (x > 3) {`	
`  myDog.play();`	
`}`	
`int[] numList = {2, 4, 6, 8};`	
`System.out.print("Hello");`	Gibt »Hello« aus … wahrscheinlich auf der Kommandozeile.
`System.out.print("Dog: " + name);`	
`String num = "8";`	
`int z = Integer.parseInt(num);`	
`try {`	
`  readTheFile("myFile.txt");`	
`}`	
`catch (FileNotFoundException ex) {`	
`  System.out.print("File not found.");`	
`}`	

F: **Die Namenskonventionen für die verschiedenen Java-Versionen sind verwirrend. Es gab JDK 1.0, 1.2, 1.3 und 1.4. Dann kam ein Sprung zu J2SE 5.0, dann wurde zu Java 6 und Java 7 gewechselt. Als ich das letzte Mal nachgesehen habe, war Java 18 die aktuelle Version. Was ist da los?**

A: Die Versionsnummern haben sich in den letzten 25 Jahren stark verändert! Die Buchstaben (J2SE/SE) können wir getrost ignorieren, da sie inzwischen kaum noch benutzt werden. Bei den Zahlen ist es etwas komplizierter.

Technisch gesehen, war Java 5.0 eigentlich Java 1.5 – und ebenso Java 6 = 1.6, 7 = 1.7 und 8 =1.8. Theoretisch hat Java immer noch die Version 1.x, weil neue Versionen rückwärtskompatibel sind bis zurück zu 1.0. Allerdings verwirrte eine Versionsnummer, die anders ist als die normalerweise genutzte. Daher ist seit Java 9 die offizielle Versionsnummer die Zahl ohne die vorangestellte 1. Java 9 ist also tatsächlich Version 9 und nicht Version 1.9.

Wir verwenden hier für die Versionen 1.0 bis 1.4 die übliche Konvention. Ab Version 5 lassen wir die vorangestellte 1 weg.

Außerdem gab es seit der Veröffentlichung von Version 9 alle sechs Monate ein neues Java-Release mit einer eigenen »Haupt«-Versionsnummer, was erklärt, warum wir in so kurzer Zeit von Version 9 zu Version 19 gelangt sind.

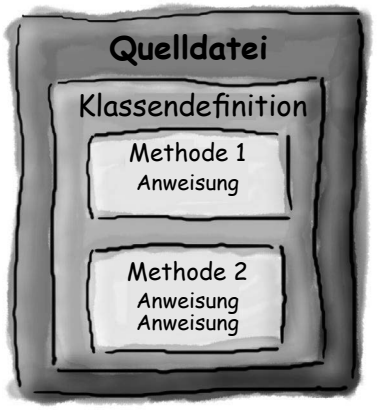
Spitzen Sie Ihren Bleistift Antworten

Sehen Sie, wie einfach es ist, Java zu schreiben

Keine Sorge, wenn Sie das hier jetzt noch nicht verstehen!
Alles, was hier steht, wird in diesem Buch ausführlich erklärt (das meiste auf den ersten 40 Seiten). Wenn Java einer Sprache ähnelt, mit der Sie schon gearbeitet haben, wird Ihnen einiges recht leichtfallen. Falls nicht, machen Sie sich keine Sorgen. *Wir schaffen das schon ...*

`int size = 27;`	Deklariert eine Integer-Variable namens »size« und weist ihr den Wert 27 zu.
`String name = "Fido";`	Deklariert einen String namens »name« und weist ihm den Wert »Fido« zu.
`Dog myDog = new Dog(name, size);`	Deklariert eine neue Dog-Variable »myDog« und bewirkt, dass der neue Hund »name« und »size« verwendet.
`x = size - 5;`	Subtrahiert 5 von 27 (dem Wert von »size«) und weist ihn der Variablen »x« zu.
`if (x < 15) myDog.bark(8);`	Ist x (mit dem Wert 22) kleiner als 15, wird der Hund (Dog) angewiesen, achtmal zu bellen.
`while (x > 3) {`	Die Schleife wird so lange durchlaufen, wie x größer als 3 ist ...
`  myDog.play();`	Sagt dem Hund (myDog), dass er spielen soll (was auch immer DAS für einen Hund bedeutet).
`}`	Das sieht nach dem Ende der Schleife aus – alles in { } wird in der Schleife ausgeführt.
`int[] numList = {2, 4, 6, 8};`	Deklariert eine Liste von Integer-Variablen namens »numList« und packt 2, 4, 6 und 8 in die Liste.
`System.out.print("Hello");`	Gibt »Hello« aus ... wahrscheinlich auf der Kommandozeile.
`System.out.print("Dog: " + name);`	Gibt auf der Kommandozeile »Dog: Fido« aus (der Wert von »name« ist Fido).
`String num = "8";`	Deklariert einen String »num« und weist ihm den Wert »8« zu.
`int z = Integer.parseInt(num);`	Wandelt den String »8« in den tatsächlichen numerischen Wert 8 um.
`try {`	Versucht, etwas zu tun ... vielleicht ist nicht sicher, dass das, was hier versucht wird, funktioniert ...
`  readTheFile("myFile.txt");`	Liest eine Textdatei namens »myFile.txt« ein (oder VERSUCHT zumindest, die Datei zu lesen).
`}`	Das muss das Ende der »zu versuchenden Dinge« sein, ich vermute mal, man kann viele Dinge versuchen.
`catch (FileNotFoundException ex) {`	Das muss die Stelle sein, an der man herausfindet, ob das, was man versucht hat, geklappt hat ...
`  System.out.print("File not found.");`	Schlägt der Versuch fehl, wird auf der Kommandozeile »File not found« ausgegeben.
`}`	Offenbar sollen die Dinge in { } getan werden, wenn 'try' nicht funktioniert hat.

Codestruktur in Java



Packen Sie eine Klasse in eine Quelldatei.

Packen Sie Methoden in eine Klasse.

Packen Sie Anweisungen in eine Methode.

Was kommt in eine Quelldatei?

Eine Quelldatei (mit der Endung *.java*) enthält typischerweise eine ***Klassendefinition***. Die Klasse stellt einen ***Teil*** Ihres Programms dar – wobei ganz winzige Anwendungen vielleicht auch nur aus einer Klasse bestehen. Die Klasse muss mit einem Paar geschweifter Klammern umgeben werden.

```
public class Dog {




}
```

Klasse

Was kommt in eine Klasse?

Eine Klasse enthält eine oder mehrere ***Methoden***. Die Methode ***bark*** (bellen) der Klasse Dog sollte Anweisungen dazu enthalten, wie der Hund bellen soll. Ihre Methoden müssen ***innerhalb*** einer Klasse (das heißt zwischen den geschweiften Klammern der Klasse) deklariert werden.

```
public class Dog {
   void bark() {



   }
}
```

Methode

Was kommt in eine Methode?

Zwischen die geschweiften Klammern einer Methode schreiben Sie die Anweisungen, die angeben, wie die Methode ausgeführt werden soll. Im Prinzip besteht der Methoden-***code*** aus einer Reihe von Anweisungen. Im Moment reicht es, wenn Sie sich eine Methode als eine Art Funktion oder Prozedur vorstellen.

```
public class Dog {
   void bark() {
     statement1;
     statement2;
   }
}
```

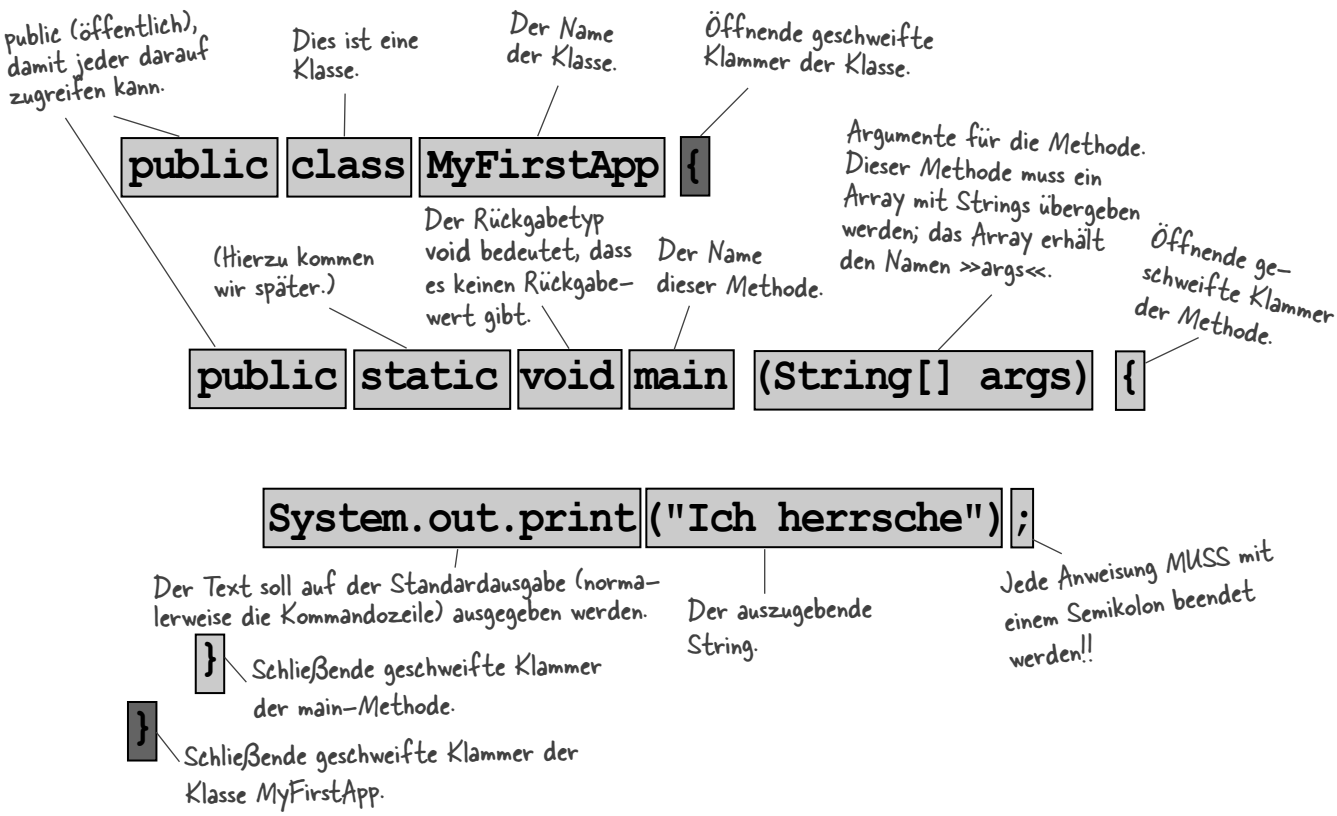
Anweisungen

Anatomie einer Java-Klasse

Wenn die JVM gestartet wird, sucht sie nach der Klasse, die Sie auf der Kommandozeile übergeben haben. Dann beginnt sie mit der Suche nach einer speziell geschriebenen Methode, die genau so aussieht:

```
public static void main (String[] args) {
   // hier steht Ihr Code
}
```

Danach führt die JVM alles zwischen den geschweiften Klammern { } Ihrer main-Methode aus. Jede Java-Applikation muss mindestens eine **Klasse** und eine **main**-Methode enthalten (nicht eine main-Methode pro *Klasse*, nur eine main-Methode pro *Anwendung*).



Machen Sie sich nicht die Mühe, sich das jetzt schon zu merken.
Dieses Kapitel soll Sie nur auf den Weg bringen.

Eine Klasse mit einer main()-Methode schreiben

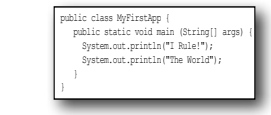
In Java steht sämtlicher Code in einer **Klasse**. Sie tippen Ihre Quelldatei (mit der Dateiendung ***.java***) ein und kompilieren diese in eine neue Klasse (mit der Endung ***.class***). Wenn Sie Ihr Programm laufen lassen, wird also tatsächlich eine Klasse ausgeführt.

Wenn Sie ein Programm ausführen, sagen Sie der Java Virtual Machine (JVM) im Prinzip: »Lade die Klasse **`MyFirstApp`** und beginne dann mit der Ausführung der enthaltenen **`main(-`**Methode. Führe das Programm so lange aus, bis der gesamte Code in main beendet ist.«

In Kapitel 2 werden wir uns diese ganze Sache mit den Klassen gründlicher ansehen. Im Augenblick reicht es, wenn Sie wissen, ***wie Sie JavaCode schreiben, damit er ausgeführt werden kann***. Und das beginnt alles mit main().

Die **main()**-Methode ist der Ort, an dem die Ausführung Ihres Programms beginnt.

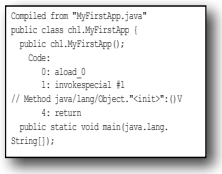
Unabhängig von der Größe Ihres Programms (das heißt von der Anzahl der *Klassen*, die Ihr Programm verwendet) muss es grundsätzlich eine **main()-Methode** geben, damit alles in Gang kommen kann.



1 Speichern

`MyFirstApp.java`

MyFirstApp.java



MyFirstApp.class

2 Kompilieren

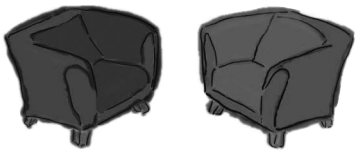
`javac MyFirstApp.java`

3 Ausführen

`java MyFirstApp`

```
public class MyFirstApp {

   public static void main (String[] args) {
     System.out.println("I Rule!");
     System.out.println("The World");
   }

}
```

Kamingespräche



Heute Abend: **Der Compiler und die JVM streiten über die Frage: »Wer ist wichtiger?«**

Die Java Virtual Machine

Wie bitte? Soll das ein Scherz sein? ***HALLO***. Ich bin Java. Ich bin der Typ, der die Programme tatsächlich zum *Laufen* bringt. Der Compiler gibt Ihnen bloß eine Datei. Das ist alles. Nur eine *Datei*. Sie können sie ausdrucken und dann an die Wand kleben, in den Kamin stecken oder damit den Vogelkäfig auslegen. Was Sie wollen. Aber die Datei macht *nichts*, wenn ich nicht da bin, um sie auszuführen.

Stimmt. Das hatte ich fast vergessen. Außerdem versteht der Compiler keinerlei Spaß. Na, kein Wunder. Wenn *Sie* sich den ganzen Tag damit herumschlagen müssten, pedantisch kleine Syntaxverletzungen zu prüfen ...

Ich habe ja nicht gesagt, Sie seien, ähm, *völlig* nutzlos. Aber was machen Sie denn eigentlich? Ernsthaft. Ich habe keine Ahnung. Ein Programmierer könnte den Bytecode doch einfach von Hand schreiben und mir geben. Nicht mehr lange, dann sind Sie arbeitslos, Kumpel.

(Ich sag's ja: kein Humor!) Aber Sie haben meine Frage noch nicht beantwortet: Was *machen* Sie eigentlich?

Der Compiler

Dieser Ton gefällt mir gar nicht.

Entschuldigung, aber ohne *mich* würde was genau ausgeführt werden? Damit Sie's wissen: Es gibt einen *Grund*, warum Java einen Bytecode-Compiler verwendet. Wäre Java nur eine interpretierte Sprache, bei der die virtuelle Maschine den Quellcode direkt aus einer Textdatei übersetzen müsste, würde ein Java-Programm nicht einmal im Schneckentempo ablaufen.

Entschuldigung! Aber diese Sichtweise ist ziemlich ignorant (um nicht zu sagen, arrogant). *Theoretisch* mag es stimmen, dass Sie jeden korrekt formatierten Bytecode ausführen können, selbst wenn er nicht aus einem Java-Compiler stammt. Trotzdem ist das in der Praxis ziemlich absurd. Von Hand Bytecode schreiben? Das ist, als würden Sie Ihre Urlaubsbilder malen, anstatt Fotos zu machen. Klar ist das Kunst, aber die meisten Menschen nutzen ihre Zeit lieber anders. Mich würde es übrigens freuen, wenn Sie darauf *verzichten*, mich als Ihren »Kumpel« zu bezeichnen.

Denken Sie daran, dass Java eine streng typisierte Sprache ist. Das heißt, dass ich nicht zulassen darf, dass Variablen Daten des falschen Typs

Die Java Virtual Machine

Aber manche kommen immer noch durch! Ich kann ClassCastExceptions auslösen, und manchmal hab ich mit Leuten zu tun, die Zeug des falschen Typs in ein Array stecken, das deklarationsgemäß anderes Zeug enthalten soll, und ...

Klar. Sicher. Aber was ist mit *Sicherheit*? Sehen Sie sich mal all den Sicherheitskrempel an, den ich mache. Und Sie, wie war das noch mal, prüfen *Semikola*? Oooohhh. Wahnsinniges Sicherheitsrisiko! Was ein Glück, dass es Sie gibt.

Ganz egal. Ich mach das gleiche Zeug auch, allerdings nur, um sicherzustellen, dass sich niemand hinter Ihnen reinschleicht und den Bytecode ändert, bevor er ausgeführt wird.

Worauf Sie sich verlassen können, *Kumpel*.

Der Compiler

enthalten. Das ist ein entscheidendes Sicherheitsfeature. Und ich kann die meisten Verstöße aufhalten, bevor Sie Ihnen jemals vorgesetzt werden. Und ich ...

Entschuldigung! Aber ich war noch nicht fertig. Ja, es *gibt* ein paar Datentyp-Exceptions, die zur Laufzeit auftreten können. Aber einige von denen wurden erlaubt, um eins der wichtigen Features von Java zu unterstützen – dynamische Bindung. Ein Java-Programm kann zur Laufzeit neue Objekte einschließen, die dem ursprünglichen Programmierer noch nicht einmal *bekannt* waren. Deswegen besitze ich einen gewissen Grad an Flexibilität. Mein Job ist es, alles aufzuhalten, was zur Laufzeit nie erfolgreich wäre – nie erfolgreich sein *könnte*. In der Regel kann ich sagen, wenn etwas nicht funktioniert. Wenn ein Programmierer aus Versehen versucht, ein Button-Objekt als Socket-Verbindung zu verwenden, würde ich das beispielsweise entdecken und ihn so davor schützen, zur Laufzeit Schaden zu verursachen.

Entschuldigung, aber ich bin die erste Verteidigungslinie, wie man so sagt. Die Datentypverletzungen, die ich eben beschrieben habe, könnten in einem Programm Chaos verursachen, wenn ihr Auftreten zugelassen würde. Außerdem bin ich der, der Zugriffsverletzungen verhindert, wie Code, der versucht, eine private Methode aufzurufen, oder eine Methode ändert, die – aus Sicherheitsgründen – nie geändert werden darf. Ich halte Menschen davon ab, Code anzurühren, den sie nicht sehen sollen, beispielsweise wenn Code versucht, auf die kritischen Daten einer anderen Klasse zuzugreifen. Es könnte Stunden, vielleicht sogar Tage dauern, wollte man die Bedeutung meiner Arbeit beschreiben.

Natürlich, aber wie ich bereits gesagt habe: Würde ich nicht annähernd 99 % der möglichen Probleme verhindern, würden Sie sich ständig festfahren. Aber jetzt ist anscheinend unsere Zeit abgelaufen. Wir müssen das also in einem späteren Gespräche noch mal ausdiskutieren.

Was kann man über die main-Methode sagen?

Sobald Sie sich in main (oder *irgendeiner* anderen Methode) befinden, fängt der Spaß an. Sie können all die üblichen Sachen sagen, die Sie in den meisten Programmiersprachen sagen können, ***um den Computer dazu zu bringen, etwas zu machen.***

Ihr Code kann der JVM Folgendes sagen:

1 Mach was!

Anweisungen: Deklarationen, Zuweisungen, Methodenaufrufe etc.

```
int x = 3;
String name = "Dirk";
x = x * 17;
System.out.print("x is " + x);
double d = Math.random();
// dies ist ein Kommentar
```

2 Mach was immer und immer wieder!

Schleifen: *for* und *while*

```
while (x > 12) {
  x = x - 1;
}

for (int i = 0; i < 10; i = i + 1) {
  System.out.print("i is now " + i);
}
```

3 Mach was unter dieser Bedingung!

Verzweigungen: *if/else*-Tests

```
if (x == 10) {
  System.out.print("x must be 10");
} else {
  System.out.print("x isn't 10");
}
if ((x < 3) && (name.equals("Dirk"))) {
  System.out.println("Gently");
}
System.out.print("this line runs no matter what");
```

Syntax-Spaß

★Jede Anweisung muss mit einem Semikolon abgeschlossen werden.

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

★Ein einzeiliger Kommentar wird durch zwei Schrägstriche eingeleitet.

```
x = 22;
// diese Zeile stört mich
```

★Die meisten Whitespace-Zeichen (Leerzeichen, Tabulator etc.) spielen keine Rolle.

```
x      =      3  ;
```

★Variablen werden mit einem **Namen** und einem **Typ** deklariert (alles Wissenswerte zu Javas *Typen* finden Sie in Kapitel 3).

```
int weight;
//Typ: int, Name: weight
```

★Klassen und Methoden müssen in einem Paar geschweifter Klammern definiert werden.

```
public void go() {
   // fantastischer Code
}
```

Schleifen, Schleifen, Schleifen ...

Java hat eine Menge Schleifenkonstrukte, von denen while, do-while und *for* die ältesten sind. Die volle Schleifenpackung kommt später im Buch. Das dauert aber noch etwas. Daher kümmern wir uns erst mal um while.

Die Syntax (von der Logik ganz zu schweigen) ist so einfach, dass Sie vermutlich schon eingeschlafen sind: Solange eine Bedingung wahr ist, mach alles, was im Schleifen*block* steht. Der Schleifenblock wird von einem Paar geschweifter Klammern umgeben. Alles, was wiederholt werden soll, muss also in diesem Block stehen.

Der Schlüssel zu einer Schleife ist die *Testbedingung*. In Java ist die Testbedingung ein Ausdruck, der einen *booleschen* Wert zum Ergebnis hat – anders ausgedrückt: etwas, das entweder ***wahr*** oder ***falsch*** ist.

Wenn Sie etwas wie »Solange *eisInSchüssel ist wahr*, machEiskugeln« sagen, haben Sie einen wunderbaren booleschen Test. Entweder *ist* Eis in der Schüssel oder es ist *keins* in der Schüssel. Wenn Sie dagegen etwas sagen wie »Solange *Bob* machEiskugeln«, haben Sie keinen echten Test. Damit das funktioniert, müssen Sie es in etwas ändern wie »Solange Bob schnarcht ...« oder »Solange Bob *nicht* trägtUmhang ...«.

Einfache boolesche Tests

Einen einfachen booleschen Test können Sie durchführen, indem Sie mit einem *Vergleichsoperator* den Wert einer Variablen prüfen. Zu diesen Operatoren zählen:

< (kleiner als)

> (größer als)

== (gleich) (ja, das sind *zwei* Gleichheitszeichen)

Beachten Sie den Unterschied zwischen dem *Zuweisungsoperator* (ein einzelnes Gleichheitszeichen) und dem *Gleichheitsoperator* (zwei Gleichheitszeichen). Viele Programmierer tippen aus Versehen **=** ein, wenn sie eigentlich **==** *haben möchten*. (Sie natürlich nicht.)

```
int x = 4; // weist x den Wert 4 zu
while (x > 3) {
  // Schleifencode wird ausgeführt, weil
  // x größer als 3 ist
  x = x - 1; // oder die Schleife läuft
             // für immer
}
int z = 27; //
while (z == 17) {
  // Schleifencode läuft nicht, weil
  // z nicht gleich 17 ist
}
```

Es gibt keine Dummen Fragen

F: **Warum muss alles in einer Klasse stehen?**

A: Java ist eine objektorientierte (OO-)Sprache. Früher, in der Zeit der dampfgetriebenen Compiler, musste man dagegen noch eine monolithische Quelldatei mit einem Haufen Prozeduren schreiben. In Kapitel 2 werden Sie lernen, dass eine Klasse eine Art Schablone für ein Objekt ist und dass in Java eigentlich fast alles ein Objekt ist.

F: **Muss jede Klasse, die ich schreibe, ein main enthalten?**

A: Nein. Ein Java-Programm kann Dutzende (oder sogar Hunderte) von Klassen verwenden, von denen vielleicht nur *eine einzige* eine main-Methode enthält – die Klasse, die die Ausführung Ihres Programms beginnt.

F: **In vielen anderen Sprachen kann ich einen booleschen Test auch mit Integer-Werten durchführen. Kann ich in Java etwas sagen wie:**

```
int x = 1;
while (x){ }
```

A: Nein. Die Typen *boolean* (boolescher Wert) und *integer* (Ganzzahl) sind in Java nicht miteinander kompatibel. Da das Ergebnis einer Testbedingung ein boolescher Wert sein muss, ist die einzige Variable, die Sie direkt testen können (ohne einen Vergleichsoperator zu verwenden), ein ***boolescher Wert***. Sie können beispielsweise sagen:

```
boolean isHot = true;
while(isHot) { }
```

Beispiel einer while-Schleife

```
public class Loopy {
  public static void main(String[] args) {
    int x = 1;
    System.out.println("Before the Loop");
    while (x < 4) {
      System.out.println("In the loop");
      System.out.println("Value of x is " + x);
      x = x + 1;
    }
    System.out.println("This is after the loop");
  }
}
```

Dies ist die Ausgabe.

```
% java Loopy
Before the Loop
In the loop
Value of x is 1
In the loop
Value of x is 2
In the loop
Value of x is 3
This is after the loop
```

PUNKT FÜR PUNKT

- Anweisungen werden mit einem Semikolon beendet: ;
- Codeblöcke werden mit einem Paar geschweifter Klammern definiert: **{ }**
- Eine *int*-Variable mit einem Namen und einem Typ deklarieren: **int x;**
- Der **Zuweisungs**operator ist *ein* Gleichheitszeichen: **=**
- Der **Gleichheits**operator verwendet *zwei* Gleichheitszeichen: **==**
- Eine *while*-Schleife führt alles in ihrem Block (definiert durch geschweifte Klammern) aus, solange die Testbedingung ***true*** (wahr) ist.
  Ist die Testbedingung ***false*** (falsch), wird der Codeblock der *while*-Schleife nicht ausgeführt. Die Ausführung geht unmittelbar mit dem Code weiter, der auf den Schleifenblock folgt:
  `while (x == 4) { }`

Bedingte Verzweigungen

In Java ist ein *if*-(falls-)Test im Wesentlichen mit einem booleschen Test in einer *while*-Schleife identisch. Nur, dass Sie nicht sagen »***Solange*** es Schokolade gibt ...«, sondern »***Falls*** es Schokolade gibt ...«.

```
class IfTest {
  public static void main (String[] args) {
    int x = 3;
    if (x == 3) {
      System.out.println("x must be 3");
    }
    System.out.println("This runs no matter what");
  }
}
```

Codeausgabe

```
% java IfTest
x must be 3
This runs no matter what
```

Der oben stehende Code führt die Zeile aus, die »x must be 3« ausgibt, allerdings nur, *falls* die Bedingung (*x* ist gleich 3) true (wahr) ist. Unabhängig davon wird die Zeile, die »This runs no matter what« ausgibt, immer ausgeführt. Abhängig vom Wert von *x* erzeugen also eine oder zwei Anweisungen eine Ausgabe.

Aber wir können die Bedingung um ein else (sonst) erweitern, um etwas auszudrücken wie: »Falls (if) es noch Schokolade gibt, schreib das Programm weiter, sonst (else) besorge mehr Schokolade und mach dann weiter mit ...«

```
class IfTest2 {
  public static void main(String[] args) {
    int x = 2;
    if (x == 3) {
      System.out.println("x must be 3");
    } else {
      System.out.println("x is NOT 3");
    }
    System.out.println("This runs no matter what");
  }
}
```

neue Ausgabe

```
% java IfTest2
x is NOT 3
This runs no matter what
```

System.out.print oder System.out.println?

Wenn Sie aufgepasst haben, ist Ihnen aufgefallen, dass wir zwischen **print** und **println** gewechselt haben.

Haben Sie den Unterschied bemerkt?

System.out.***println*** fügt am Zeilenende einen Zeilenumbruch (newline) ein (denken Sie sich print***ln*** als **print*new*line**), bei System.out.***print*** wird die Ausgabe dagegen auf der gleichen Zeile fortgesetzt. Soll jede Ausgabe auf einer neuen Zeile stattfinden, verwenden Sie print**ln**. Soll alles auf einer gemeinsamen Zeile stehen, nutzen Sie print.

Spitzen Sie Ihren Bleistift

Für folgende Ausgabe:

```
% java DooBee
DooBeeDooBeeDo
```

ergänzen Sie den fehlenden Code:

```
public class DooBee {
  public static void main(String[] args) {
    int x = 1;
    while (x < _____ ) {
      System.out._________("Doo");
      System.out._________("Bee");
      x = x + 1;
    }
    if (x == _______  ) {
      System.out.print("Do");
    }
  }
}
```

Antworten auf Seite 25.

Eine ernsthafte Geschäftsanwendung schreiben

Setzen wir unsere neu erworbenen Java-Fähigkeiten mal für etwas Praktisches ein. Wir brauchen eine Klasse mit einem *main()*, einem *int* und einer *String*-Variablen, einer *while*-Schleife und einem *if*-Test. Das Ganze noch etwas aufpolieren, und im Handumdrehen haben Sie das Business-Backend fertig. Aber bevor Sie sich den Code auf dieser Seite betrachten, denken Sie einen Moment darüber nach, wie Sie den Code für den alten Kinderklassiker »10 green bottles« schreiben würden.

```
public class BottleSong {
  public static void main(String[] args) {
    int bottlesNum = 10;
    String word = "bottles";

    while (bottlesNum > 0) {

      if (bottlesNum == 1) {
        word = "bottle"; // Singular, wie in EINE Flasche (one bottle).
      }

      System.out.println(bottlesNum + " green " + word + ", hanging on the wall");
      System.out.println(bottlesNum + " green " + word + ", hanging on the wall");
      System.out.println("And if one green bottle should accidentally fall,");
      bottlesNum = bottlesNum - 1;

      if (bottlesNum > 0) {
        System.out.println("There'll be " + bottlesNum +
                           " green " + word + ", hanging on the wall");
      } else {
        System.out.println("There'll be no green bottles, hanging on the wall");
      } // Ende else
    } // Ende der while-Schleife
  } // Ende main
} // Ende class
```

Unser Code hat noch einen kleinen Haken. Er lässt sich kompilieren und ausführen, aber die Ausgabe ist nicht vollkommen perfekt. Schauen Sie mal, ob Sie das Problem finden und reparieren können.

Es gibt keine Dummen Fragen

F: Hieß das nicht früher mal »99 Bierflaschen«?

A: Ja, aber Trisha wollte unbedingt die britische Version des Liedes nehmen. Wenn Sie die Version mit den 99 Flaschen vorziehen, dürfen Sie die Übung gerne umschreiben.

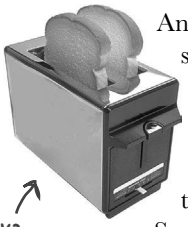
Montagmorgen in Bobs Java-fähigem Haus

Wie an jedem Wochentag klingelt Bobs Wecker um 8:30 Uhr. Bob hatte allerdings ein ziemlich wildes Wochenende und drückt die SCHLUMMERN-Taste. Und das ist der Moment, in dem die Action losgeht und seine Java-fähigen Geräte zum Leben erwachen ...

Zuerst schickt der Wecker eine Nachricht an die Kaffeemaschine: »Hey, der Geek verpennt schon wieder. Warte mit dem Kaffee noch 12 Minuten.«

Die Kaffeemaschine schickt darauf eine Nachricht an den Java-Toaster: »Warte mit dem Toast, Bob schnarcht noch.«

Dann schickt der Wecker eine Nachricht an Bobs Android: »Bob um Punkt 9 Uhr anrufen und ihm sagen, dass wir ein bisschen spät dran sind.«



Schließlich schickt der Wecker eine Nachricht an Sams WLAN-Halsband (Sam ist der Hund) mit dem so vertrauten Signal: »Hol die Zeitung, aber mach dir keine Hoffnung auf einen Spaziergang.«

Ein paar Minuten später klingelt der Wecker schon wieder, und Bob drückt erneut auf SCHLUMMERN. Darauf beginnen die Geräte noch einmal, miteinander zu reden. Schließlich klingelt der Wecker ein drittes Mal. Aber während Bob noch den SCHLUMMERN-Knopf sucht, schickt der Wecker ein »Spring auf und bell!«-Signal an Sams Halsband. Erschreckt und jetzt hellwach steht Bob auf, dankbar für seine Java-Fähigkeiten und die spontanen Interneteinkäufe, die ihm seinen Alltag so sehr vereinfacht haben.

Sein Toast ist getoastet.

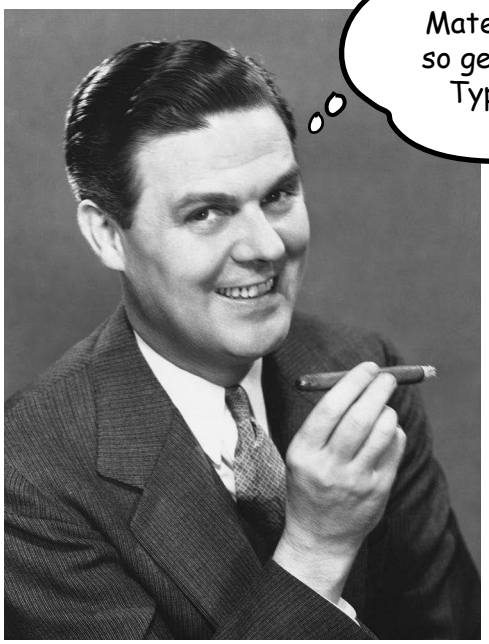
Sein Kaffee dampft.

Seine Zeitung wartet.

Ein weiterer wunderschöner Morgen in Bobs ***Java-fähigem Haus***.

Könnte diese Geschichte wahr sein? Größtenteils ja! Es gibt Versionen von Java, die auf verschiedenen Endgeräten laufen, inklusive Handys (besonders auf Handys), Geldautomaten, Kreditkarten, Haussicherheitssystemen, Parkuhren, Videospiel-Controllern und vielem mehr – nur noch nicht auf Hundehalsbändern. Das ist aber nur eine Frage der Zeit.

Mit Java ist es möglich, nur eine winzige Untermenge der Java-Plattform auf Kleingeräten laufen zu lassen (je nachdem, welche Java-Version Sie verwenden), was gerade für die IoT-Entwicklung (Internet of Things) sehr beliebt ist. Und natürlich wird ein großer Teil der Android-Entwicklung mit Java und JVM-Sprachen durchgeführt.



Na gut, das Bier-Lied war keine wirklich *ernsthafte* Geschäftsanwendung. Brauchen Sie immer noch etwas Praktisches, das Sie Ihrem Boss vorzeigen können? Dann sehen Sie sich mal den Phras-O-Mat-Code an.

Hinweis: Wenn Sie das in einen Editor eingeben, lassen Sie den Editor die Wort-/Zeilenumbrüche steuern! Drücken Sie nie die Enter-Taste, während Sie einen String (das Zeug zwischen den "Anführungszeichen") eingeben. Der Code lässt sich dann nicht mehr kompilieren. Die Bindestriche, die Sie auf dieser Seite sehen, sind echt und können eingegeben werden. Aber Enter dürfen Sie erst drücken, NACHDEM Sie einen String geschlossen haben.

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

    // Drei Wortgruppen erstellen, aus denen ausgewählt wird.
    // Fügen Sie Ihre eigenen hinzu!
    String[] wordListOne = {"agnostic", "opinionated",
"voice activated", "haptically driven", "extensible",
"reactive", "agent based", "functional", "AI enabled",
"strongly typed"};

    String[] wordListTwo = {"loosely coupled", "six sigma",
"asynchronous", "event driven", "pub-sub", "IoT", "cloud
native", "service oriented", "containerized", "serverless",
"microservices", "distributed ledger"};

    String[] wordListThree = {"framework", "library",
"DSL", "REST API", "repository", "pipeline", "service
mesh", "architecture", "perspective", "design",
"orientation"};

    // Herausfinden, wie viele Wörter die Listen enthalten.
    int oneLength = wordListOne.length;
    int twoLength = wordListTwo.length;
    int threeLength = wordListThree.length;

    // Drei Zufallszahlen erzeugen.
    java.util.Random randomGenerator = new java.util.Random();
    int rand1 = randomGenerator.nextInt(oneLength);
    int rand2 = randomGenerator.nextInt(twoLength);
    int rand3 = randomGenerator.nextInt(threeLength);

    // Eine neue Phrase zusammensetzen.
    String phrase = wordListOne[rand1] + " " +
wordListTwo[rand2] + " " + wordListThree[rand3];

    // Die Phrase ausgeben.
    System.out.println("What we need is a " + phrase);
  }
}
```

Der Phras-O-Mat

Wie er funktioniert

Kurz gesagt: Das Programm erstellt drei Listen mit Wörtern, wählt dann zufällig jeweils ein Wort aus jeder der drei Listen und gibt die drei Wörter hintereinander aus. Machen Sie sich keine Gedanken, wenn Sie nicht *genau* verstehen, was in allen Zeilen passiert. Meine Güte, Sie haben schließlich noch das ganze Buch vor sich. Entspannen Sie sich einfach. Dies ist nur ein kurzer Blick auf eine »AI enabled cloud native architecture«.

Was wir brauchen, ist eine ...

extensible microservices pipeline

opinionated loosely coupled REST API

agent-based microservices library

AI-enabled service oriented orientation

functional IoT perspective

1. Im ersten Schritt werden drei String-Arrays erzeugt – die Container, die alle Wörter enthalten. Die Deklaration und Erzeugung eines Arrays ist einfach. Hier ist ein kleines:

```
String[] pets = {"Fido", "Zeus", "Bin"};
```

Die einzelnen Wörter stehen (wie alle braven String) in Anführungszeichen und werden durch Kommata getrennt.

2. Aus jeder der drei Listen (Arrays) soll zufällig ein Wort ausgewählt werden. Also müssen wir wissen, wie viele Wörter die einzelnen Listen enthalten. Wenn eine Liste14 Wörter enthält, müssen wir eine Zufallszahl zwischen 0 und 13 erzeugen (Java-Arrays sind nullbasiert, das erste Element befindet sich also an Position 0, das zweite an Position 1 und das letzte bei einem Array mit 14 Elementen an Position 13). Praktischerweise ist es Java-Arrays ein Vergnügen, Ihnen ihre Länge mitzuteilen. Sie müssen sie einfach nur fragen. Beim Array pets würden wir Folgendes sagen:

```
int x = pets.length;
```

x würde dann den Wert 3 enthalten.

3. Wir brauchen drei Zufallszahlen. Java besitzt standardmäßig bereits mehrere Möglichkeiten, Zufallszahlen zu erzeugen, inklusive java.util.Random (wir werden später sehen, warum diesem Klassennamen ein java.util vorangestellt wird). Die **`nextInt()`**-Methode gibt eine zufällige Zahl zwischen 0 und der an sie übergebenen Zahl zurück, *exklusive* der übergebenen Zahl selbst. Also übergeben wir ihr die Anzahl der Elemente (die Länge des Arrays) der gerade verwendeten Liste. Danach weisen wir die einzelnen Ergebnisse einer neuen Variablen zu. Wir hätten auch genauso gut nach einer Zufallszahl zwischen 0 und 5 ohne die 5 selbst fragen können:

```
int x = randomGenerator.nextInt(5);
```

4. Jetzt bauen wir die Phrase zusammen, indem wir aus allen drei Listen je ein Wort auswählen und diese zusammenpappen (und dazwischen jeweils ein Leerzeichen einfügen). Hierfür benutzen wir den »**+**«-Operator, der die String-Objekte *konkateniert* (wir bevorzugen den technischeren Ausdruck *zusammenpappen*). Um ein Element aus dem Array zu bekommen, geben Sie dem Array die Indexnummer (Position) des Dings, das Sie benutzen wollen:

```
String s = pets[0]; // s ist jetzt der String "Fido"
s = s + " " + "is a dog"; // s ist jetzt der String "Fido is a dog"
```

5. Zum Schluss geben wir die Phrase auf der Kommandozeile aus und ... voilà! *Wir sind im Marketing.*

Codemagnete

Ein funktionierendes Java-Programm ist vollkommen durcheinandergeraten. Können Sie die Codeschnipsel wieder so anordnen, dass sie ein funktionierendes Java-Programm ergeben, das die unten aufgeführte Ausgabe erzeugt? Einige der geschweiften Klammern sind zu Boden gefallen. Sie waren einfach zu klein, um sie aufzuheben. Fügen Sie so viele davon ein, wie Sie benötigen!

```
if (x == 1) {
    System.out.print("d");
    x = x - 1;
}
```

```
if (x == 2) {
    System.out.print("b c");
}
```

```
class Shuffle1 {
  public static void main(String [] args) {
```

```
if (x > 2) {
    System.out.print("a");
}
```

```
int x = 3;
```

```
x = x - 1;
System.out.print("-");
```

```
while (x > 0) {
```

Ausgabe:

```
Datei Bearbeiten Fenster Hilfe Schlafen
% java Shuffle1
a-b c-d
```

→ Antworten auf Seite 25.

SEIEN Sie der Compiler

Jede der Java-Dateien auf dieser Seite stellt eine vollständige Quelldatei dar. Spielen Sie Compiler und ermitteln Sie, welche der Dateien kompiliert wird. Wenn eine Datei nicht kompiliert wird, wie würden Sie sie reparieren?

→ Antworten auf Seite 25.

A

```
class Exercise1a {
  public static void main(String[] args) {
    int x = 1;
    while (x < 10) {
      if (x > 3) {
        System.out.println("big x");
      }
    }
  }
}
```

B

```
public static void main(String [] args) {
  int x = 5;
  while ( x > 1 ) {
    x = x - 1;
    if ( x < 3) {
      System.out.println("small x");
    }
  }
}
```

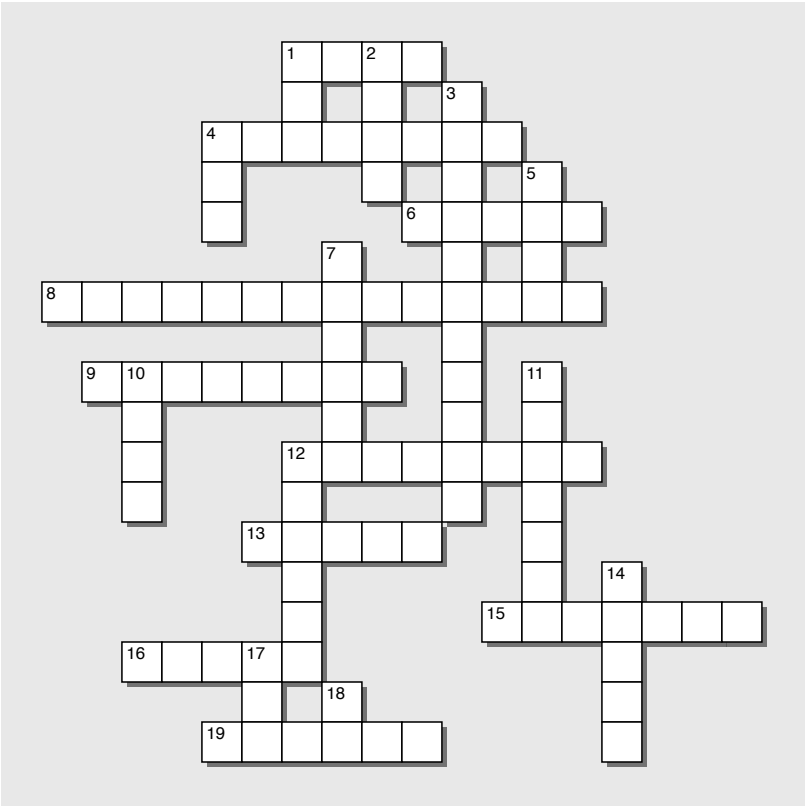
C

```
class Exercise1c {
  int x = 5;
  while (x > 1) {
    x = x - 1;
    if (x < 3) {
      System.out.println("small x");
    }
  }
}
```

Java-Kreuzworträtsel

Jetzt ist Ihre rechte Hirnhälfte auch mal dran.

Dies ist ein gewöhnliches Kreuzworträtsel, aber fast alle Lösungswörter stammen aus Kapitel 1. Damit Sie nicht einschlafen, haben wir ein paar (Nicht-Java-)Wörter aus der Hightechwelt eingestreut.



Waagerecht

1. Kommandozeilenaufrufer.
4. Quellcode-Schlucker.
6. Kein Integer.
8. Sagt etwas.
9. Kann man nicht festnageln.
12. Manchmal ist einmal nicht genug.
13. »Dinge«-Halter.
15. Hiermit tun die Dinge etwas.
16. Kann nicht in beide Richtungen gehen.
19. Schockierender Modifizierer.

Senkrecht

1. Bytecode-Schlucker.
2. Kommt mit leeren Händen zurück.
3. Eine neue Klasse oder Methode ankündigen.
4. Akronym für Prozessor.
5. Muss man nur einmal haben.
7. Offenes Haus.
10. Unzuverlässigste Laptop-Komponente.
11. Wofür eine Eingabeaufforderung gut ist.
12. Eine ordentliche Mannschaft von Zeichen.
14. Solange die Bedingung zufrieden ist.
17. Variablentyp für Zahlen.
18. Die Abteilung, ohne die nichts geht.

→ Antworten auf Seite 26.

Vermischte Nachrichten

Unten sehen Sie ein kurzes Java-Programm, bei dem ein Block fehlt. Ihre Herausforderung besteht darin, **die möglichen Codeblöcke** (links) **den Ausgaben zuzuordnen**, die Sie sehen würden, wenn der Block eingefügt würde. Nicht alle Ausgabezeilen werden verwendet, und einige Zeilen mit Ausgaben können mehrfach eingesetzt werden. Verbinden Sie die Codeblockkandidaten mit den passenden Kommandozeilenausgaben. (Die Antworten finden Sie am Ende des Kapitels.)

```
class Test {
  public static void main(String [] args) {
    int x = 0;
    int y = 0;
    while (x < 5) {
      // Hier kommt der Kandidatencode hin.
      System.out.print(x + "" + y +" ");
      x = x + 1;
    }
  }
}
```

Ordnen Sie jedem Kandidaten eine der möglichen Ausgaben zu.

Kandidaten:

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if( y > 4 ) {
  y = y - 1;
}
```

```
x = x + 1;
y = y + x;
```

```
if ( y < 5 ) {
  x = x + 1;
  if ( y < 3 ) {
    x = x - 1;
  }
}
y = y + 2;
```

Mögliche Ausgaben:

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

→ Antworten auf Seite 26.

Pool-Puzzle

Ihre ***Aufgabe*** ist es, die Codeschnipsel aus dem Pool zu fischen und sie auf den leeren Zeilen im Code zu platzieren. Jeder Schnipsel darf nur einmal verwendet werden, und Sie werden nicht alle Schnipsel brauchen. Ihr ***Ziel*** ist es, eine Klasse zu erstellen, die kompiliert und ausgeführt wird, um die gezeigte Ausgabe zu erzeugen. Lassen Sie sich nicht täuschen. Dieses Rätsel ist schwerer, als es aussieht.

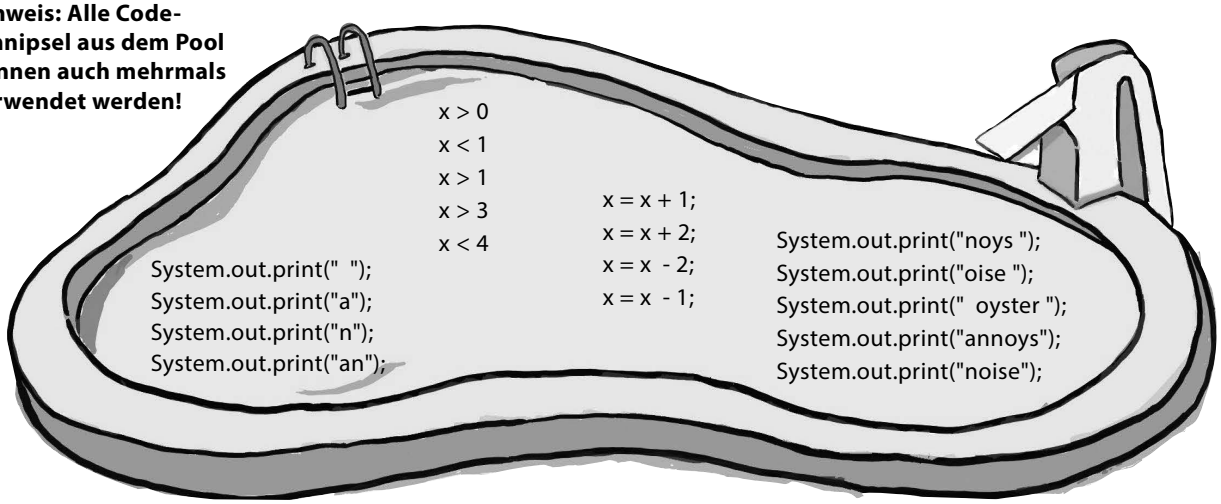
→ Antworten auf Seite 26.

Ausgabe

```
Datei Bearbeiten Fenster Hilfe Schummeln
%java PoolPuzzleOne
a noise
annoys
an oyster
```

```
class PoolPuzzleOne {
  public static void main(String [] args) {
    int x = 0;

    while ( ___________ ) {

      _____________________________
      if ( x < 1 ) {
        _____________________________
      }
      _____________________________

      if ( ___________ ) {

        _____________________________

        ___________
      }
      if ( x == 1 ) {

        _____________________________
      }
      if ( ___________ ) {

        _____________________________
      }
      System.out.println();

      ___________
    }
  }
}
```

Hinweis: Alle Codeschnipsel aus dem Pool können auch mehrmals verwendet werden!



Lösungen zu den Übungen

Spitzen Sie Ihren Bleistift (von Seite 15)

```
public class DooBee {
  public static void main(String[] args) {
    int x = 1;
    while (x < 3) {
      System.out.print("Doo");
      System.out.print("Bee");
      x = x + 1;
    }
    if (x == 3) {
      System.out.print("Do");
    }
  }
}
```

Codemagnete (von Seite 20)

```
class Shuffle1 {
  public static void main(String[] args) {

    int x = 3;
    while (x > 0) {

      if (x > 2) {
        System.out.print("a");
      }

      x = x - 1;
      System.out.print("-");

      if (x == 2) {
        System.out.print("b c");
      }

      if (x == 1) {
        System.out.print("d");
        x = x - 1;
      }
    }
  }
}
```

```
Datei Bearbeiten Fenster Hilfe Dichten
% java Shuffle1
a-b c-d
```

SEIEN Sie der Compiler (von Seite 21)

A

```
class Exercise1a {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      x = x + 1;          <- Fügen Sie diese Zeile hinzu, damit
      if ( x > 3) {          das Programm nicht ewig läuft ...
        System.out.println("big x");
      }
    }
  }
}
```

Dieser Code wird kompiliert und ausgeführt (ohne Ausgaben), aber ohne die zusätzliche Zeile würde das Programm in einer unendlichen while-Schleife ewig weiterlaufen!

B

```
class Exercise1b {   <- Braucht eine Klassendeklaration.
  public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
  }
}
```

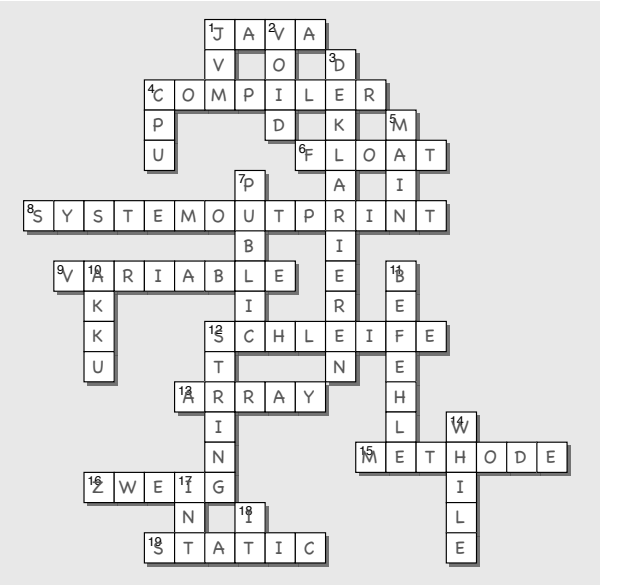
Diese Datei würde ohne Klassendeklaration nicht kompiliert werden. Und vergessen Sie nicht die passende geschweifte Klammer!

C

```
class Exercise1c {   <- Braucht ein >>main<<.
  public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
  }
}
```

Die while-Schleife muss sich innerhalb einer Methode befinden. Sie kann nicht einfach irgendwo in der Klasse herumhängen.

Java-Kreuzworträtsel (von Seite 22)



Pool-Puzzle (von Seite 24)

```
class PoolPuzzleOne {
  public static void main(String [] args) {
    int x = 0;

    while ( x<4 ) {

      System.out.print("a");
      if ( x < 1 ) {
        System.out.print(" ");
      }
      System.out.print("n");

      if ( x>1 ) {
        System.out.print(" oyster");
        x=x+2;
      }
      if ( x == 1 ) {
        System.out.print("noys");
      }
      if ( x<1 ) {
        System.out.print("oise");
      }
      System.out.println();

      x=x+1;
    }
  }
}
```

```
Datei Bearbeiten Fenster Hilfe Schummeln
%java PoolPuzzleOne
a noise
annoys
an oyster
```

Vermischte Nachrichten

```
class Test {
  public static void main(String [] args) {
    int x = 0;
    int y = 0;
    while ( x < 5 ) {
      [          ]
      System.out.print(x + "" + y +" ");
      x = x + 1;
    }
  }
}
```

Kandidaten:

```
y = x - y;

y = y + x;

y = y + 2;
if( y > 4 ) {
  y = y - 1;
}

x = x + 1;
y = y + x;

if ( y < 5 ) {
  x = x + 1;
  if ( y < 3 ) {
    x = x - 1;
  }
}
y = y + 2;
```

Mögliche Ausgaben:

```
22 46

11 34 59

02 14 26 38

02 14 36 48

00 11 21 32 42

11 21 32 42 53

00 11 23 36 410

02 14 25 36 47
```

(von Seite 23)

Die Reise nach Objectville

Ich dachte, hier gibt's Objekte. In Kapitel 1 haben wir den ganzen Code in die main()-Methode gepackt. Das ist aber nicht besonders objektorientiert – eigentlich überhaupt nicht. Na gut, wir haben ein paar Objekte benutzt, wie die String-Arrays für den Phras-O-Maten, aber eigene Objekttypen haben wir eigentlich nicht entwickelt. Aber jetzt ist endlich die Zeit gekommen, die prozedurale Welt hinter uns zu lassen. Nichts wie raus aus main() und ran an die Erstellung unserer eigenen Objekte! Wir werden erfahren, warum die objektorientierte Entwicklung in Java so viel Spaß macht. Außerdem werfen wir einen Blick auf den Unterschied zwischen einer Klasse und einem Objekt und darauf, wie Objekte Ihr Leben verbessern können (zumindest Ihr Programmiererleben – für Ihren Lifestyle sind Sie selbst verantwortlich). Aber Vorsicht! Eine Reise nach Objectville ist meistens eine Reise ohne Wiederkehr. Schicken Sie uns eine Postkarte.

Stuhlkriege

(oder: wie Objekte Ihr Leben verändern)

Es war einmal in einer Softwarefirma: Einer Programmiererin und einem Programmierer wurde die gleiche Spezifikation in die Hand gedrückt, die sie »realisieren« sollten. Der echt nervige Projektmanager zwang die beiden Entwickler zu einem Wettbewerb. Wer zuerst abliefert, bekommt einen dieser coolen Bürostühle und einen höhenverstellbaren Schreibtisch, wie ihn all die Typen aus dem Silicon Valley haben. Laura, die prozedurale Programmiererin, und Brad, der OO-Entwickler, wussten beide, dass das ein Kinderspiel werden würde.

Laura saß an ihrem (nicht verstellbaren) Schreibtisch und fragte sich: »Welche Dinge soll dieses Programm *tun*? Welche ***Prozeduren*** werden dafür gebraucht?« Und sie gab selbst die Antwort: »**Rotieren** und **Sound abspielen**.« Und schon machte sie sich auf, die Prozedur zu schreiben. Was *ist* denn ein Programm anderes als ein Haufen von Prozeduren?

Währenddessen hing Brad im Café rum und dachte: »Um welche ***Dinge*** geht es in diesem Programm, wer sind die ***Hauptakteure***?« Erst dachte er an **die Formen**, dann fielen ihm aber noch andere Dinge ein: die Benutzer, der Klang, das Click-Event. Aber für diese Dinge hatte er bereits eine Codebibliothek. Also konzentrierte er sich auf die Programmierung der Formen. Lesen Sie weiter, um zu sehen, wie Laura und Brad ihre Programme geschrieben haben, und erfahren Sie die Antwort auf die brennende Frage: »***Wer bekam am Ende den coolen Bürostuhl und den Schreibtisch***?«

der Bürostuhl

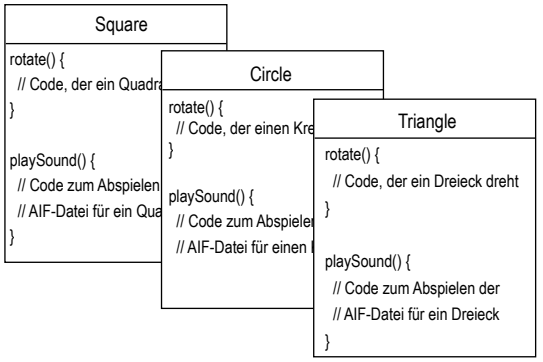
An Lauras Schreibtisch

Wie schon unzählige Male zuvor begann Laura, ihre **wichtigen Prozeduren** zu schreiben. Mit **rotate** (drehen) und **playSound** (Ton abspielen) war sie im Handumdrehen fertig.

```
rotate(shapeNum) {
  // die Form um 360° drehen
}
playSound(shapeNum) {
  // verwendet shapeNum, um heraus-
  // zufinden, welcher AIF-Ton
  // abgespielt werden
  // soll, und spielt ihn ab
}
```

Auf Brads Laptop im Café

Brad schrieb für jede der drei Formen eine ***Klasse***:



Laura dachte, sie hätte alles im Griff. Schon spürte sie den Stuhl unter ihrem ...

Moment! Die Spezifikation hat sich geändert.

»Gut, *technisch* gesehen, waren Sie die Erste, Laura«, sagte der Manager, »aber wir müssen dem Programm noch eine winzige Kleinigkeit hinzufügen. Für zwei Programmierprofis wie Sie beide ist das sicher kein Problem, oder?«

»*Bekäme ich jedes Mal, wenn ich das höre, einen Cent ...*«, dachte Laura bei sich, denn sie wusste, dass problemlose Änderungen der Spezifikation nur ein Hirngespinst waren. »*Aber wieso bleibt Brad dabei so ruhig? Was ist denn da los?*« Laura blieb bei ihrer Grundüberzeugung, dass der OO-Weg zwar ganz nett, aber einfach nur langsam ist. Und wenn man diese Überzeugung ändern wollte, müsste man sie schon aus ihren kalten, toten, von Sehnenscheidenentzündung gezeichneten Klauen reißen.

Was der Spezifikation hinzugefügt wurde.

Zurück an Lauras Schreibtisch

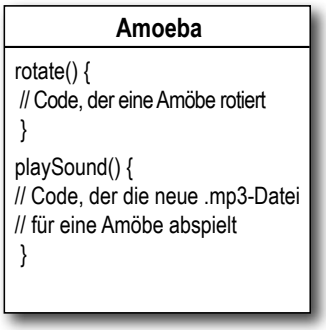
Die Prozedur zum Drehen (rotate) würde weiter funktionieren. Der Code verwendete eine Lookup-Tabelle, um die Nummer einer Form (shapeNum) der tatsächlichen Grafik einer Form zuzuordnen. ***Aber* playSound (Ton abspielen) *müsste angepasst werden***.

```
playSound(shapeNum) {
  // falls die Form keine Amöbe ist,
    // verwende shapeNum, um nachzusehen, welche
    // AIF-Sounddatei abgespielt werden soll,
  // sonst
    // spiele die Amöben-.mp3-Sounddatei ab.
}
```

Am Ende war das gar nicht so schlimm. Ein ***mulmiges Gefühl kam dennoch in ihr hoch, als sie ihren schon getesteten Code noch einmal anfassen musste***. Gerade sie sollte wissen, dass sich ***die Spezifikation immer ändert***, egal was der Projektmanager sagt.

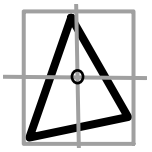
Auf Brads Laptop am Strand

Brad grinste, nippte an seiner Margerita und *schrieb noch eine neue Klasse*. Es gab Zeiten, in denen er OO besonders liebte, da er bereits getesteten und ausgelieferten Code nicht mehr anrühren musste. »Flexibilität, Erweiterbarkeit ...«, dachte er, sich die Vorteile von OO ins Gedächtnis rufend.



Laura hat nur knapp vor Brad abgegeben.

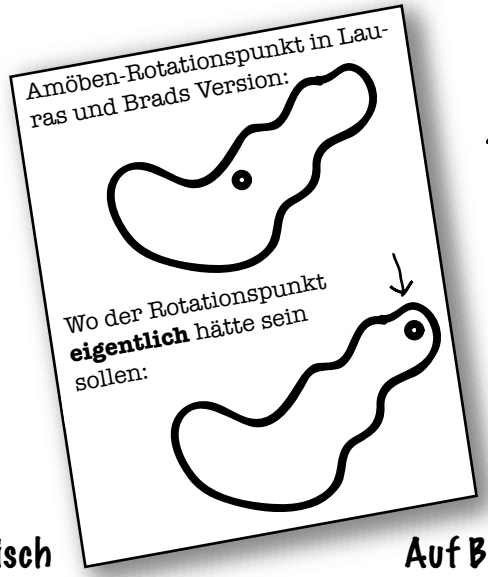
(Hah! So viel zu dem unausgegorenen OO-Quatsch!) Das Grinsen verging Laura aber schnell wieder, als der sehr nervige Projektmanager (in einem Ton aufgesetzter Enttäuschung) sagte: »Ah. Nein. *So* sollte die Amöbe nicht rotieren ...«



Wie sich zeigte, hatten beide Programmierer Code zum Rotieren nach diesem Schema geschrieben:

1. **Berechne das umgebende Rechteck für die Form.**
2. **Berechne das Zentrum des Rechtecks und drehe die Form um diesen Punkt.**

Eigentlich sollte die Amöbenform aber wie ein Uhrzeiger um einen Punkt an einem Ende rotieren. *»Jetzt bin ich geliefert«,* dachte Laura und fühlte sich dabei selbst ein bisschen wie eine Amöbe. »Obwohl ..., ich könnte einfach noch eine if/else-Anweisung in die Rotationsanweisung einbauen und den Rotationspunkt für die Amöbe dann hartcodieren. Dabei wird schon nichts kaputtgehen.« Aber die Stimme in Kopf sagte: »*Großer Fehler. Glaubst du wirklich, die Spezifikation wird sich danach nicht mehr ändern?*«



Was die Spezifikation freundlicherweise vergessen hatte zu erwähnen.

Zurück an Lauras Schreibtisch

Laura entschied, dass es besser sei, die Prozedur für das Drehen um Argumente für den Rotationspunkt zu erweitern. ***Eine Menge Code war betroffen***. Testen, neu kompilieren, und zwar alles immer und immer wieder. Dinge, die zuvor funktionierten, taten das jetzt nicht mehr.

```
rotate(shapeNum, xPt, yPt) {
  // falls die Form keine Amöbe ist,
    // berechne das Zentrum
    // basierend auf dem Rechteck,
    // dann drehe,
  // sonst
    // benutze die Koordinaten
    // xPt und yPt als Versatz des
    // Rotationspunkts und drehe dann
}
```

Auf Brads Laptop auf seinem Campingstuhl beim Telluride Bluegrass Festival

Ohne einen Takt zu verpassen, modifizierte Brad die rotate-**Methode** – aber nur in der Amoeba-Klasse. ***Den bereits getesteten, funktionierenden und kompilierten Code für die übrigen Programmteile ließ er unangetastet***. Um die Amöbe mit einem Rotationspunkt zu versehen, fügte er ein **Attribut** hinzu, das alle Amöben haben werden. Er passte den Code an, testete ihn und lieferte das überarbeitete Programm (über das kostenlose Festival-WLAN) während eines einzelnen Sets von Jerry Douglas ab.

Amoeba

```
int xPoint
int yPoint
rotate() {
// Code zum Rotieren einer Amöbe
// auf Basis der x- und y-Werte
}
playSound() {
// Code, der die neue .mp3-Datei
// für eine Amöbe abspielt
 }
```

Das heißt, der OO-Typ Brad hat den Stuhl bekommen?

Nicht so schnell. Laura hat einen Makel in Brads Ansatz gefunden. Sie war sich sicher: Bekäme sie Stuhl und Tisch, wäre auch eine Beförderung nicht mehr weit. Sie musste dieser Sache also unbedingt noch eine Wendung geben.

LAURA: Du hast duplizierten Code: Die rotate-Prozedur taucht in allen vier Shape-Dingern auf.

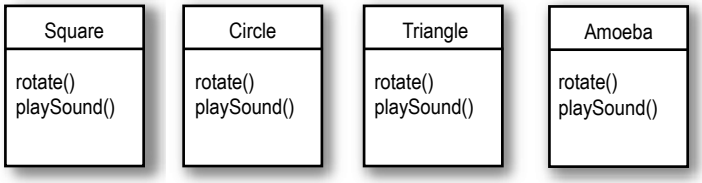
BRAD: Das ist eine ***Methode***, keine *Prozedur*. Und das sind auch keine *Dinger*, sondern ***Klassen***,

LAURA: Ja egal. Das Design ist jedenfalls Käse. Du musst *vier* verschiedene rotate-»Methoden« pflegen. Was soll daran gut sein?

BRAD: Oh, ich glaube, Du hast das fertige Design noch gar nicht gesehen. Komm, ich zeige dir mal, wie OO-**Vererbung** funktioniert, Laura.

Was Laura eigentlich wollte

(Sie dachte, der Stuhl würde sie einer Beförderung und dem großen Geld näher bringen.)

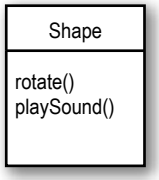


1

Ich habe nach dem gesucht, was alle vier Klassen gemeinsam haben.

2

Wir haben vier Formen (Shapes). Alle sollen gedreht werden und einen Sound abspielen können. Also habe ich die gemeinsamen Merkmale abstrahiert und in eine neue Klasse namens Shape (Form) gepackt.

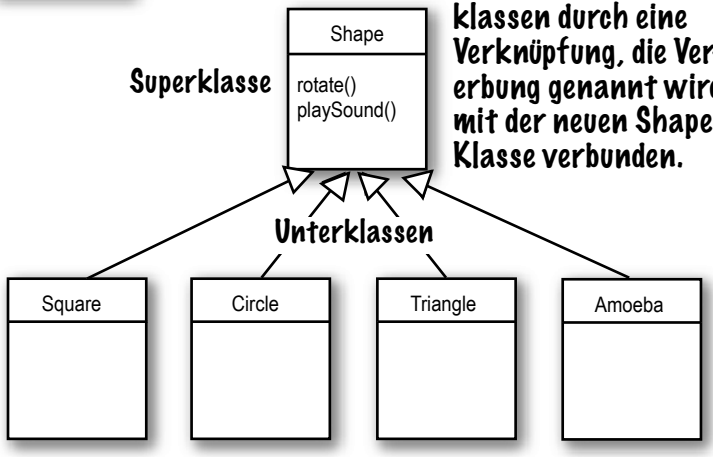


Das können Sie lesen als: **»Square (Quadrat) erbt von Shape (Form)«, »Circle (Kreis) erbt von Shape«** und so weiter. Ich habe die Methoden rotate() und playSound() aus den anderen Formen entfernt, wodurch nur noch eine Kopie gepflegt werden muss.

Die Shape-Klasse nennt man **Superklasse** der anderen vier Klassen. Andersherum bezeichnet man die anderen vier als **Unterklassen (oder Subklassen)** von Shape. Die Unterklassen *erben* die Methoden von der Superklasse. Anders gesagt: *Durch die Vererbung ist die Funktionalität der Shape-Klasse automatisch in den Unterklassen nutzbar.*

3

Danach habe ich die anderen vier Formenklassen durch eine Verknüpfung, die Vererbung genannt wird, mit der neuen Shape-Klasse verbunden.



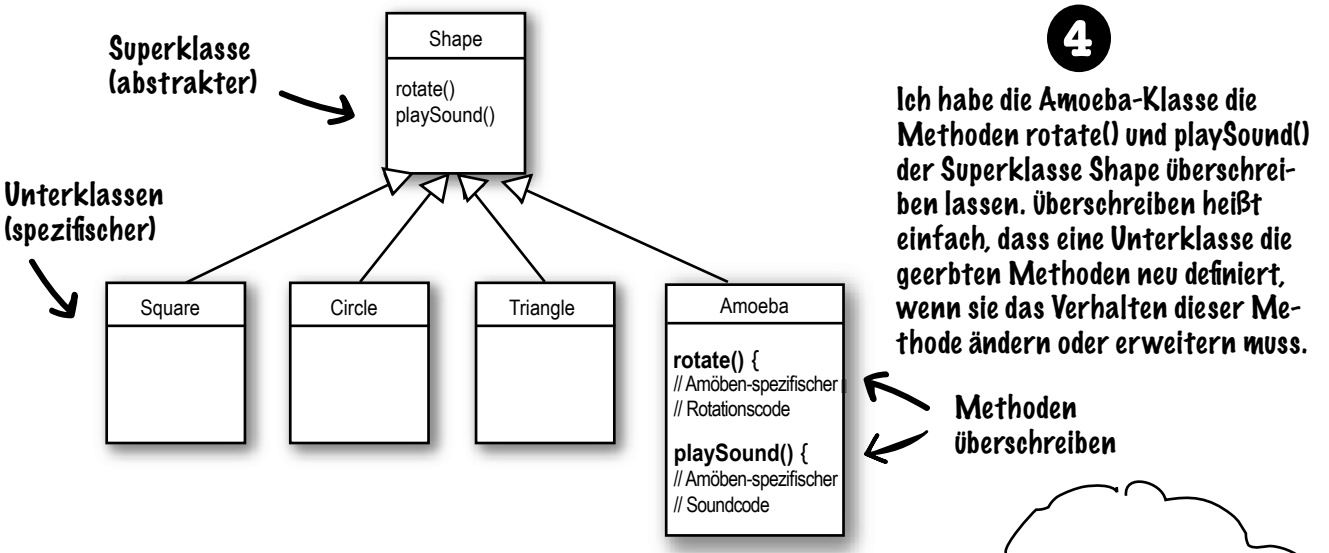
Was ist mit der rotate()-Methode für die Amöbenform?

LAURA: War das nicht gerade das Problem, dass die Amöbenform eine komplett andere rotate- und playSound-Prozedur brauchte?

BRAD: Methode.

LAURA: Ja ja. Wie kann sich die Amoeba-Klasse anders verhalten, wenn sie ihre Funktionalität von der Shape-Klasse erbt?

BRAD: Das ist der letzte Schritt. Die Amoeba-Klasse **überschreibt** die Methode der Shape-Klasse. Zur Laufzeit weiß die JVM dann genau, welche rotate()-Methode ausgeführt werden muss, wenn jemand die Amöbenform anweist, sich zu drehen.

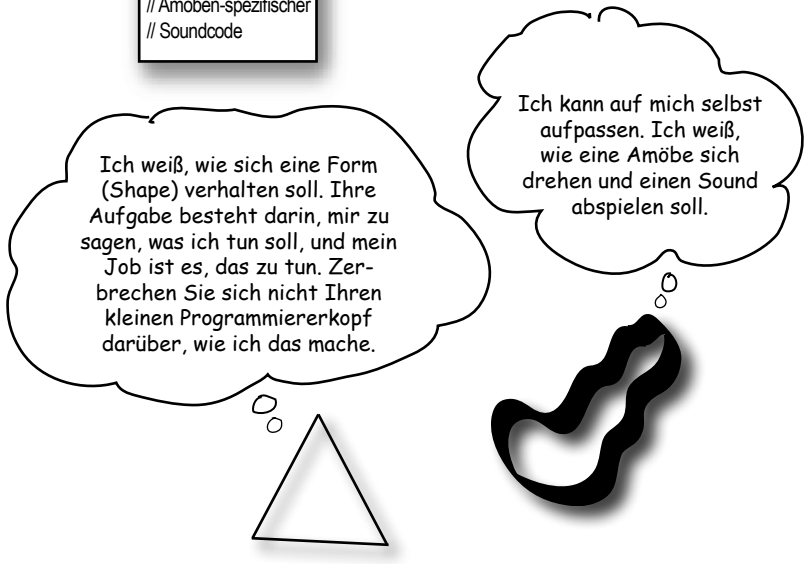


4

Ich habe die Amoeba-Klasse die Methoden rotate() und playSound() der Superklasse Shape überschreiben lassen. Überschreiben heißt einfach, dass eine Unterklasse die geerbten Methoden neu definiert, wenn sie das Verhalten dieser Methode ändern oder erweitern muss.

LAURA: Wie kannst du einer Amöbenform »sagen«, was sie tun soll? Musst du nicht die Prozedur aufrufen – sorry, die Methode – und ihr dann sagen, *was* gedreht werden soll?

BRAD: Das ist ja gerade das Coole an OO. Wenn es zum Beispiel für ein Dreieck Zeit ist, sich zu drehen, ruft der Programmcode die rotate()-Methode *auf dem Triangle-Objekt* auf. Dem Rest des Programms ist dabei vollkommen egal, *wie* das Dreieck das macht. Und wenn du das Programm erweitern musst, schreibst du einfach eine neue Klasse für den neuen Objekttyp, **damit die neuen Objekte ihr eigenes Verhalten bekommen.**



Die Spannung bringt mich noch um. Also, wer hat den Stuhl denn jetzt bekommen?

Amy aus dem zweiten Stock.

(Ohne es jemandem zu sagen, hat der Projektmanager die Spezifikation an drei Programmiererinnen und Programmierer gegeben. Amy war am schnellsten mit dem Projekt fertig, weil sie sich mit OO-Programmierung auskennt und außerdem keine Zeit damit verschwendete, mit ihren Kollegen herumzustreiten.)

Was gefällt dir an OO?

»OO hilft mir, meine Entwürfe auf natürlichere Weise zu gestalten. Solche Dinge stecken immer in einem Entwicklungsprozess.«

– Sabine, 27, Softwarearchitektin

»Ich muss nicht an Code rumbasteln, den ich bereits getestet habe, sondern kann einfach ein neues Feature hinzufügen.«

– Brad, 32, Programmierer

»Mir gefällt, dass die Daten und die Methoden, die auf diesen Daten operieren, in einer Klasse zusammengehalten werden.«

–John, 22, Biertrinker

»Dass man Code in anderen Anwendungen wiederverwenden kann. Wenn ich eine neue Klasse schreibe, kann ich sie so flexibel machen, dass sie später in etwas Neuem verwendet werden kann.«

– Chris, 39, Projektmanager

»Unglaublich, dass ausgerechnet Chris das sagen muss. Er hat seit fünf Jahren keine einzige Zeile Code mehr geschrieben.«

– Daryl, 44, arbeitet für Chris

»Abgesehen von dem Stuhl?«

– Amy, 34, Programmiererin

KOPF-NUSS

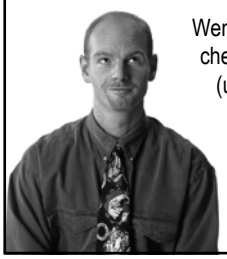
Zeit, ein paar Neuronen zu massieren.

Sie haben gerade eine Geschichte über eine prozedurale Programmiererin gelesen, die sich mit einem OO-Programmierer messen musste. Sie haben einen kurzen Überblick über einige OO-Schlüsselkonzepte erhalten, wie etwa Klassen und Methoden. Das restliche Kapitel werden wir damit verbringen, uns Klassen und Objekte anzusehen (zur Vererbung und zum Überschreiben werden wir in späteren Kapiteln zurückkommen).

Berücksichtigen Sie, was Sie bisher gesehen haben (und was Sie eventuell aus früheren Arbeiten mit anderen OO-Sprachen wissen), und nehmen Sie sich einen Moment Zeit, um über die folgenden Fragen nachzudenken:

Auf welche grundsätzlichen Dinge müssen Sie beim Design einer Java-Klasse achten? Welche Fragen sollten Sie sich dabei stellen? Welche Punkte würden Sie in eine Checkliste für den Entwurf einer neuen Klasse aufnehmen?

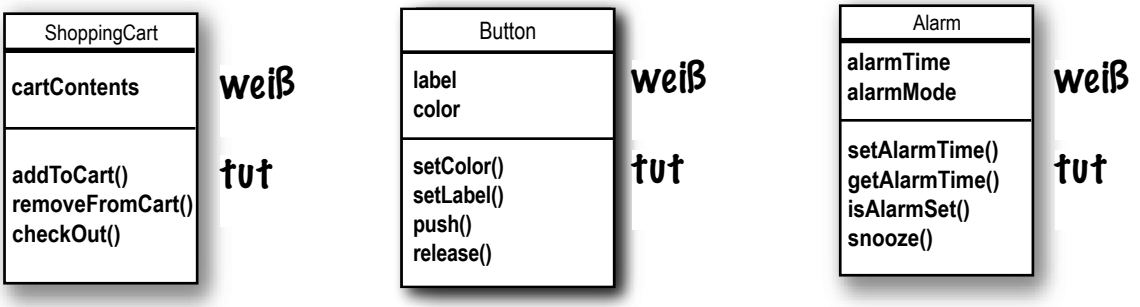
Metakognitions-Tipp



Wenn Sie bei einer Übung feststecken, versuchen Sie, laut darüber zu sprechen. Sprechen (und hören) aktiviert ein anderes Hirnareal. Am besten funktioniert das, wenn Sie das Problem mit einer anderen Person diskutieren können, aber Haustiere gehen auch. So hat unser Hund gelernt, was Polymorphie ist.

Wenn Sie eine Klasse entwerfen, sollten Sie an die Objekte denken, die von dieser Klasse erzeugt werden. Denken Sie an:

- Dinge, die das Objekt **weiß**
- Dinge, die das Objekt **tut**

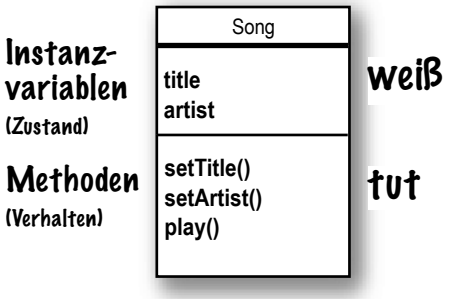


Dinge, die ein Objekt über sich selbst *weiß*, heißen

- Instanzvariablen

Dinge, die ein Objekt *tun* kann, heißen

- Methoden



Dinge, die ein Objekt über sich selbst ***weiß,*** heißen **Instanzvariablen**. Sie stehen für den Zustand (die Daten) eines Objekts und können für jedes Objekt dieses Typs unterschiedliche Werte haben.

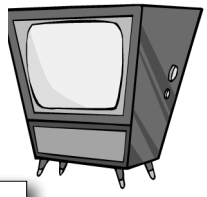
Betrachten Sie Instanz als eine andere Möglichkeit, Objekt zu sagen.

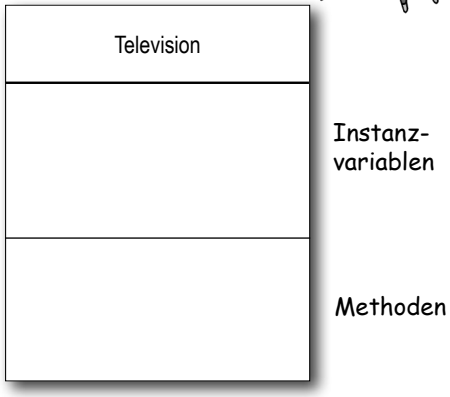
Dinge, die ein Objekt ***tun*** kann, heißen **Methoden**. Wenn Sie eine Klasse entwerfen, überlegen Sie, welche Daten ein Objekt über sich selbst wissen muss, und entwickeln Methoden, die auf diesen Daten operieren. Üblicherweise besitzt ein Objekt Methoden, die die Werte der Instanzvariablen lesen und schreiben können. So besitzen Alarm-(Wecker-)Objekte beispielsweise eine Instanzvariable, die die Weckzeit (alarmTime) enthält, und zwei Methoden, um alarmTime auszulesen (get) oder zu speichern (set).

Objekte haben also Instanzvariablen und Methoden. Diese Instanzvariablen und Methoden werden aber als Teil der Klasse entworfen.

Spitzen Sie Ihren Bleistift

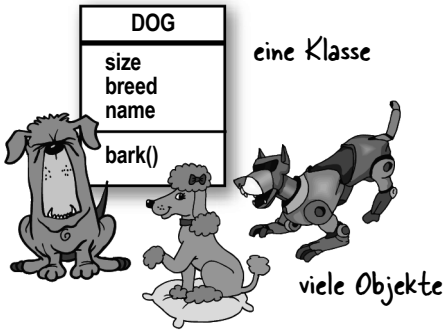
Schreiben Sie auf, was ein Fernseher-(Television-)Objekt machen und wissen muss.



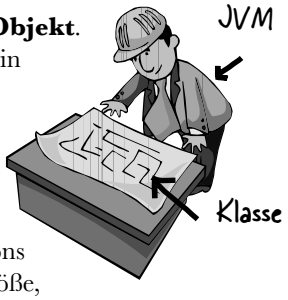


→ Diese Lösung finden Sie allein.

Was ist der Unterschied zwischen einer Klasse und einem Objekt?

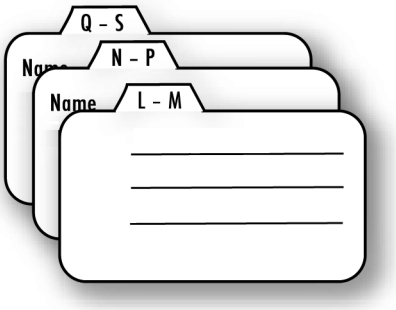


Eine Klasse ist kein Objekt, aber sie wird benutzt, um Objekte zu erzeugen.



Eine Klasse ist ein *Bauplan* für ein Objekt. Sie teilt der virtuellen Maschine mit, wie ein Objekt dieses bestimmten Typs hergestellt wird. Dabei kann jedes aus dieser Klasse erstellte Objekt eigene Werte für die Instanzvariablen der Klasse haben. Sie können beispielsweise die Button-Klasse nutzen, um Dutzende verschiedener Buttons zu erstellen, die alle eine eigene Farbe, Größe, Form, Beschriftung und so weiter haben. Und trotzdem wären alle diese Buttons Button-*Objekte*.

Sehen Sie es mal so ...



Ein Objekt ist wie ein Eintrag in Ihrem Adressbuch.

Eine Analogie für Klassen und Objekte ist das Adressbuch in Ihrem Telefon. Jeder Kontakt hat die gleichen Felder, zum Beispiel für Name, Telefonnummer, E-Mail-Adresse. Dies sind die Instanzvariablen. Die tatsächlichen Einträge entsprechen dem Zustand des Kontakts.

Die Methoden der Klasse sind die Dinge, die Sie mit einem bestimmten Kontakt machen: getName( ), changeName( ), setName( ) (Name auslesen, ändern, schreiben) könnten alle Methoden der Klasse Contact sein.

Jeder Kontakt kann also die gleichen Dinge *tun* (getName( ), changeName( ) etc.). Aber jeder einzelne Kontakt *weiß* Dinge, die nur für diesen einen Kontakt gelten.

Ihr erstes Objekt erstellen

Also, was ist nötig, um ein Objekt zu erstellen und zu benutzen? *Zwei* Klassen. Eine Klasse, die festlegt, welcher Typ Objekt benutzt werden soll (Dog, AlarmClock, Television etc.), und eine weitere Klasse, um Ihre neue Klasse zu testen. In diese *Testklasse* kommt Ihre main-Methode, und in der main()-Methode können Sie Objekte Ihres neuen Klassentyps erzeugen und auf sie zugreifen. Die Testklasse hat nur eine Aufgabe: die Methoden und Variablen Ihres neuen Objekts *auszuprobieren*.

Ab diesem Punkt werden Sie im Buch in vielen unserer Beispiele zwei Klassen sehen. Eine ist die richtige Klasse – die Klasse, deren Objekte Sie tatsächlich benutzen wollen –, die andere ist die Testklasse, die wir *<IhrXBeliebigerKlassenname>* **`TestDrive`** nennen. Wenn wir beispielsweise eine **`Bungee`**-Klasse erstellen, brauchen wir auch eine Klasse namens **`BungeeTestDrive.`** Dabei enthält nur die Klasse *<IhrXBeliebigerKlassenname>* **`TestDrive`** die main()-Methode. Ihr einziger Lebensinhalt besteht darin, Objekte Ihrer neuen Klasse (der Nicht-Testklasse) zu erstellen und dann den Punktoperator (.) zu benutzen, um auf die Methoden und Variablen des neuen Objekts zuzugreifen. Das wird anhand der folgenden Beispiele erstaunlich gut deutlich gemacht. Doch, *wirklich*.

Der Punktoperator (.)

Mit dem Punktoperator (.) können Sie auf den Zustand und das Verhalten (Instanzvariablen und Methoden) eines Objekts zugreifen.

```
// ein neues Objekt erstellen
Dog d = new Dog();

// den Hund (Dog) bellen
// lassen, indem der Punkt-
// operator auf der
// Variablen d angewendet wird,
// um bark() aufzurufen
d.bark();

// die Größe mithilfe des
// Punktoperators auf den
// Wert 40 einstellen
d.size = 40;
```

1 Schreiben Sie Ihre Klasse.

```
class Dog {
  int size;
  String breed;
  String name;

  void bark() {
    System.out.println("Ruff! Ruff!");
  }
}
```

Instanzvariablen.

Eine Methode.

2 Schreiben Sie eine Testklasse (TestDrive).

Nur eine main-Methode (der Code folgt im nächsten Schritt).

```
class DogTestDrive {
  public static void main(String[] args) {
    // hier steht der Testcode für Dog
  }
}
```

3 Erstellen Sie in Ihrer Testklasse ein Objekt und greifen Sie auf seine Variablen und Methoden zu.

```
class DogTestDrive {
  public static void main(String[] args) {
    Dog d = new Dog();
    d.size = 40;
    d.bark();
  }
}
```

Ein Dog-Objekt erstellen.

Punkt-operator.

Den Punktoperator (.) benutzen, um die Größe des Dogs einzustellen

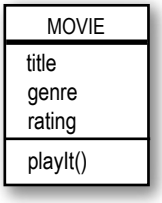
und um seine bark()-Methode aufzurufen.

Wenn Sie bereits OO-Kenntnisse haben, wissen Sie, dass wir keine Kapselung verwenden. Dazu werden wir in Kapitel 4 kommen.

Die Film-Objekte erstellen und testen


```
class Movie {
  String title;
  String genre;
  int rating;

  void playIt() {
    System.out.println("Playing the movie");
  }
}

public class MovieTestDrive {
  public static void main(String[] args) {
    Movie one = new Movie();
    one.title = "Gone with the Stock";
    one.genre = "Tragic";
    one.rating = -2;
    Movie two = new Movie();
    two.title = "Lost in Cubicle Space";
    two.genre = "Comedy";
    two.rating = 5;
    two.playIt();
    Movie three = new Movie();
    three.title = "Byte Club";
    three.genre = "Tragic but ultimately uplifting";
    three.rating = 127;
  }
}
```

Spitzen Sie Ihren Bleistift



Die Klasse MovieTestDrive erstellt Objekte (Instanzen) der Klasse Movie (Film) und benutzt den Punktoperator, um die Instanzvariablen mit einem bestimmten Wert zu versehen. MovieTestDrive ruft außerdem eine Methode an einem der Objekte auf. Füllen Sie das Diagramm auf der rechten Seite mit den Werten, die die drei Objekte am Ende von main() haben.

→ Diese Lösung finden Sie allein.

Objekt 1
- title
- genre
- rating

Objekt 2
- title
- genre
- rating

Objekt 3
- title
- genre
- rating

Schnell! Nichts wie raus aus main()!

Solange Sie sich noch in main() befinden, sind Sie nicht richtig in Objectville angekommen. Es ist in Ordnung, ein Testprogramm in main() laufen zu lassen, aber in einer echten OO-Applikation müssen Objekte mit anderen Objekten sprechen anstatt mit einer statischen main()-Methode, die Objekte erzeugt und testet.

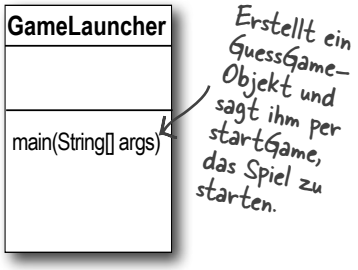
Die zwei Verwendungszwecke von main:

- **Ihre richtigen Klassen testen**
- **Ihre Java-Applikation starten**

>>Raten<< Sie mal, was >>to guess<< heißt ...

Eine echte Java-Applikation besteht aus nichts anderem als aus Objekten, die mit anderen Objekten sprechen. In diesem Fall bedeutet *sprechen*, dass Objekte Methoden aufeinander aufrufen. Auf der vorherigen Seite und in Kapitel 4 sehen wir, wie eine main()-Methode aus einer separaten TestDrive-Klasse verwendet wird, um Methoden und Variablen einer anderen Klasse zu erstellen und zu testen. In Kapitel 6 betrachten wir eine Klasse mit einer main()-Methode, die bei einer *richtigen* Java-Anwendung den Stein ins Rollen bringt (indem sie Objekte erstellt und diese Objekte dann loslässt, damit sie mit anderen Objekten interagieren etc.).

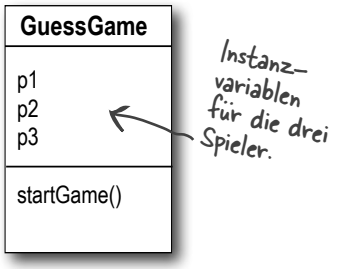
Als kleiner Ausblick auf das Verhalten einer echten Java-Applikation zeigen wir Ihnen hier ein kleines Beispiel. Da wir beim Lernen von Java immer noch ganz am Anfang stehen, werden wir nur einen kleinen Satz an Werkzeugen einsetzen. Dadurch wirkt das Programm vielleicht etwas rumpelig und ineffizient. Sie können sich ja überlegen, wie Sie das Programm verbessern. Genau das werden wir in den späteren Kapiteln übrigens auch tun. Keine Sorge, wenn einige Teile des Codes Sie verwirren. Dieses Beispiel soll hauptsächlich zeigen, wie sich Objekte miteinander unterhalten.



Das Ratespiel

Zusammenfassung:

Für unser Ratespiel (Guessing Game) sind ein Spiel-Objekt und drei Spieler-Objekte nötig. Das Spiel erzeugt eine Zufallszahl zwischen 0 und 9, und die drei Spieler müssen versuchen, die Zahl zu erraten. (Wir haben nicht behauptet, dass das Spiel besonders *aufregend* ist.)



Klassen:

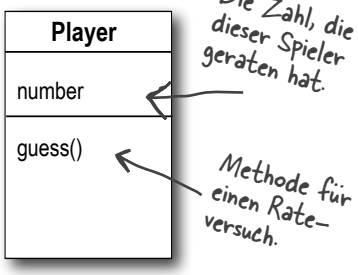
`GuessGame.class` `Player.class` `GameLauncher.class`

Die Logik:

1. In der GameLauncher-(Spiel-Starter-)Klasse beginnt die Applikation. Sie enthält die main()-Methode.

2. In der main()-Methode wird ein GuessGame-Objekt erstellt, und dessen startGame()-Methode (Spiel beginnen) wird aufgerufen.

3. In der startGame()-Methode des GuessGame-Objekts wird das vollständige Spiel abgewickelt. Sie erzeugt die drei Spieler und »denkt« sich eine Zufallszahl aus (die von den Spielern erraten werden soll). Dann fordert sie jeden Spieler auf zu raten, prüft die Ergebnisse und gibt dann entweder Informationen zu dem/den erfolgreichen Spieler(n) aus oder fordert sie auf, erneut zu raten.



```
public class GuessGame {
  Player p1;
  Player p2;
  Player p3;

  public void startGame() {
    p1 = new Player();
    p2 = new Player();
    p3 = new Player();

    int guessp1 = 0;
    int guessp2 = 0;
    int guessp3 = 0;

    boolean p1isRight = false;
    boolean p2isRight = false;
    boolean p3isRight = false;

    int targetNumber = (int) (Math.random() * 10);
    System.out.println("I'm thinking of a number between 0 and 9 ...");

    while (true) {
      System.out.println("Number to guess is " + targetNumber);

      p1.guess();
      p2.guess();
      p3.guess();

      guessp1 = p1.number;
      System.out.println("Player one guessed " + guessp1);

      guessp2 = p2.number;
      System.out.println("Player two guessed " + guessp2);

      guessp3 = p3.number;
      System.out.println("Player three guessed " + guessp3);

      if (guessp1 == targetNumber) {
        p1isRight = true;
      }
      if (guessp2 == targetNumber) {
        p2isRight = true;
      }
      if (guessp3 == targetNumber) {
        p3isRight = true;
      }

      if (p1isRight || p2isRight || p3isRight) {

        System.out.println("We have a winner!");
        System.out.println("Player one got it right? " + p1isRight);
        System.out.println("Player two got it right? " + p2isRight);
        System.out.println("Player three got it right? " + p3isRight);
        System.out.println("Game is over.");
        break; // Spiel zu Ende, Schleife verlassen
      } else {
        // Wir müssen weitermachen, weil keiner der Spielenden richtig geraten hat!
        System.out.println("Players will have to try again.");
      } // Ende von if/else
    } // Ende der while-Schleife
  } // Ende der startGame-Methode
} // Ende von class
```

GuessGame besitzt drei Instanzvariablen für die drei Player-Objekte.

Erzeugt drei Player-Objekte und weist sie den drei Spieler-Instanzvariablen (p1, p2 und p3) zu.

Deklariert drei Variablen, in denen die drei Rateversuche der Spieler gespeichert werden.

Deklariert drei Variablen, die, abhängig von der Antwort des Spielers, true oder false enthalten.

Erzeugt eine >>Zielzahl<<, die die Spieler erraten sollen (target).

Ruft die guess()-(raten-)Methode der drei Spieler auf.

Ruft die Rateversuche jedes Spielers (das Ergebnis der Ausführung ihrer guess()-Methode) ab, indem auf die number-Variable jedes Spielers zugegriffen wird.

Prüft die Rateversuche aller Spieler, um festzustellen, ob einer der Zielzahl entspricht; hat ein Spieler die richtige Zahl erraten, wird die Variable dieses Spielers auf true gesetzt (standardmäßig hat sie den Wert false).

Falls Spieler 1 ODER Spieler 2 ODER Spieler 3 recht hat (der ||-Operator bedeutet ODER).

Andernfalls in der Schleife bleiben und die Spieler zu einem neuen Rateversuch auffordern.

Das Ratespiel ausführen

```
public class Player {
int number = 0; // hier kommt der Rate-
                // versuch rein
public void guess() {
number = (int) (Math.random() * 10);
System.out.println("I'm guessing "
+ number);
}
}
public class GameLauncher {
public static void main (String[] args) {
GuessGame game = new GuessGame();
game.startGame();
}
}
```

Java bringt den Müll raus

Immer wenn in Java ein Objekt erzeugt wird, landet es in einem Speicherbereich, der als **Heap** bezeichnet wird. Alle Objekte leben im Heap, egal wann, wo oder wie sie erzeugt wurden. Das ist nicht einfach irgendein oller Speicher-Heap, tatsächlich heißt er **Garbage Collectible Heap** (auf Deutsch ungefähr »Müllsammelhaufen«). Wenn Sie ein Objekt erstellen, weist Java dem Objekt ganz nach seinen Bedürfnissen eine bestimmte Menge Speicher auf dem Heap zu. Ein Objekt mit, sagen wir mal, 15 Instanzvariablen braucht wahrscheinlich mehr Platz als ein Objekt mit nur zwei Instanzvariablen. Was passiert aber, wenn Sie den Speicherplatz wieder freigeben möchten? Wie bekommen Sie ein Objekt wieder vom Heap herunter, wenn Sie mit ihm fertig sind? Java verwaltet den Speicher für Sie! Wenn die JVM »sieht«, dass ein Objekt nie wieder benutzt werden kann, wird das Objekt *zur Garbage Collection* (Müllabfuhr) *freigegeben*. Reicht der Arbeitsspeicher nicht mehr aus, wird der Garbage Collector ausgeführt. Er schmeißt die unerreichbaren Objekte raus und gibt ihren Platz zur Wiederverwendung frei. Weiter unten im Buch zeigen wir genauer, wie das funktioniert.

Ausgabe (ist bei jeder Ausführung anders)

```
Datei Bearbeiten Fenster Hilfe Explodieren
%java GameLauncher
I'm thinking of a number between 0 and 9
...
Number to guess is 7
I'm guessing 1
I'm guessing 9
I'm guessing 9
Player one guessed 1
Player two guessed 9
Player three guessed 9
Players will have to try again.
Number to guess is 7
I'm guessing 3
I'm guessing 0
I'm guessing 9
Player one guessed 3
Player two guessed 0
Player three guessed 9
Players will have to try again.
Number to guess is 7
I'm guessing 7
I'm guessing 5
I'm guessing 0
Player one guessed 7
Player two guessed 5
Player three guessed 0
We have a winner!
Player one got it right? true
Player two got it right? false
Player three got it right? false
Game is over.
```

Es gibt keine Dummen Fragen

F: Was mache ich, wenn ich globale Variablen brauche? Wie mache ich das, wenn alles in einer Klasse stehen muss?

A: In Java-OO-Programmen gibt es das Konzept »globaler« Variablen nicht. In der Praxis kann es aber passieren, dass eine Methode (oder Konstante) für jeglichen Code an beliebiger Stelle in Ihrem Code verfügbar sein muss. Denken Sie zum Beispiel an die Methode `random()` in der Phras-O-Mat-App. Diese Methode sollte von überall aufrufbar sein. Oder wie sieht es mit Konstanten wie *Pi* aus? In Kapitel 10 werden Sie lernen, dass Methoden, die als `public` (öffentlich) und `static` (statisch) markiert sind, sich fast wie »globale« Methoden verhalten. Jeder Code in jeder Klasse Ihrer Applikation kann auf eine öffentliche und statische Methode zugreifen. Und wenn Sie eine Variable als `public`, `static` und `final` markieren, haben Sie im Prinzip eine global verfügbare *Konstante*.

F: Und was ist daran jetzt objektorientiert, wenn man trotzdem globale Funktionen und Daten anlegen kann?

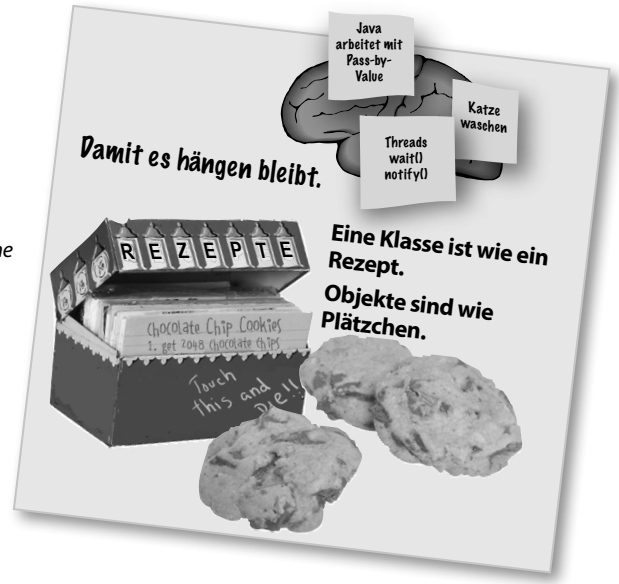
A: Zunächst einmal kommt in Java alles in eine Klasse. Die Konstante für Pi und die Methode für `random()` werden beispielsweise in der Klasse `Math` definiert, auch wenn beide public und static sind. Und Sie dürfen nicht vergessen, dass diese öffentlichen und statischen (globalartigen) Dinge in Java eher die Ausnahme als die Regel sind. Sie stehen für einen ganz besonderen Fall, in dem es nur eine Instanz/ein Objekt gibt.

F: Was *ist* ein Java-Programm? Was wird denn nun *wirklich* geliefert?

A: Ein Java-Programm ist einfach ein Haufen Klassen (oder zumindest *eine* Klasse). In einer Java-Applikation muss *eine* der Klassen eine main()-Methode enthalten, mit der das Programm gestartet wird. Als Programmierer schreiben Sie also eine oder mehrere Klassen. Und diese Klassen sind das, was Sie liefern. Haben die Endbenutzer keine JVM, müssen Sie die Ihrer Applikation auch noch beilegen, damit man Ihr Programm ausführen kann. Es gibt eine Reihe von Installationsprogrammen, mit denen Sie Ihre Klassen mit einer JVM bündeln können. Das ermöglicht Ihnen, einen Ordner oder eine Datei zu erstellen, die Sie beliebig weitergeben können (zum Beispiel über das Internet). Danach kann der Endbenutzer die korrekte Version der beiliegenden JVM installieren (sofern sie auf seinem Gerät nicht sowieso schon installiert ist).

F: Was mache ich, wenn ich 100 Klassen habe? Oder 1.000? Es muss doch furchtbar sein, diese Dateien alle einzeln auszuliefern? Kann ich das nicht irgendwie zu einem *Anwendungspaket* zusammenfassen?

A: Ja, es wäre wirklich mühsam, eine große Zahl von Einzeldateien an die Endbenutzer auszuliefern. Aber das muss nicht sein. Sie können alle Ihre Anwendungsdateien in ein Java-Archiv – eine ***.jar***-Datei – packen, die auf dem pkzip-Format basiert. Der .jar-Datei können Sie eine einfache Textdatei hinzufügen, die als sogenanntes *Manifest* formatiert ist. Sie definiert, welche Klasse in der .jar-Datei die auszuführende main()-Methode enthält.



PUNKT FÜR PUNKT

- Bei der objektorientierten Programmierung können Sie ein Programm erweitern, ohne schon getesteten und funktionierenden Code erneut anfassen zu müssen.
- Sämtlicher Java-Code wird in einer **Klasse** definiert.
- Eine Klasse beschreibt, wie ein Objekt dieses Klassentyps erstellt wird. **Eine Klasse gleicht einem Bauplan.**
- Ein Objekt kann auf sich selbst aufpassen. Sie müssen weder wissen noch sich Sorgen darum machen, *wie* das Objekt das schafft.
- Ein Objekt weiß Dinge und tut Dinge.
- Dinge, die ein Objekt über sich selbst weiß, heißen **Instanzvariablen**. Sie beschreiben den *Zustand* eines Objekts.
- Dinge, die ein Objekt tut, werden **Methoden** genannt. Sie stehen für das *Verhalten* eines Objekts.
- Wenn Sie eine Klasse erstellen, sollten Sie eventuell eine separate Testklasse erstellen, mit der Sie Objekte Ihres neuen Klassentyps erzeugen.
- Eine Klasse kann Instanzvariablen und Methoden von einer abstrakteren **Superklasse erben**.
- Zur Laufzeit besteht ein Java-Programm einfach nur aus Objekten, die sich mit anderen Objekten »unterhalten«.

SEIEN Sie der Compiler

Jede der Java-Dateien auf dieser Seite steht für eine komplette Quelldatei. Spielen Sie Compiler und finden Sie heraus, ob jede dieser Dateien kompiliert wird. Falls nicht, wie würden Sie sie reparieren? Und wenn sie kompiliert werden: Was wären ihre Ausgaben?

A

```
class StreamingSong {

  String title;
  String artist;
  int duration;

  void play() {
    System.out.println("Playing song");
  }

  void printDetails() {
    System.out.println("This is " + title +
                       " by " + artist);
  }
}

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

    song.artist = "The Beatles";
    song.title = "Come Together";
    song.play();
    song.printDetails();
  }
}
```

B

```
class Episode {

  int seriesNumber;
  int episodeNumber;

  void skipIntro() {
    System.out.println("Skipping intro ...");
  }

  void skipToNext() {
    System.out.println("Loading next episode ...");
  }
}

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

    Episode episode = new Episode();
    episode.seriesNumber = 4;
    episode.play();
    episode.skipIntro();
  }
}
```

→ Antworten auf Seite 46.