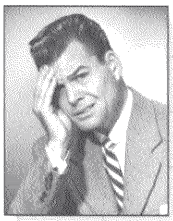
Ein Buch zum Mitmachen und Verstehen

*Behandelt Java 5.0*

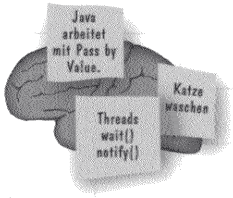
# Java™ von Kopf bis Fuß

Erfahren Sie, wie Threads Ihr Leben verändern können



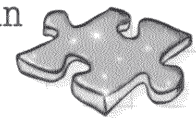
Vermeiden Sie peinliche OO-Fehler

Sorgen Sie dafür, dass die Java-Konzepte auch wirklich in Ihrem Hirn haften bleiben




Stöbern Sie in der Java-Bibliothek herum

Trainieren Sie Ihren Kopf mit 42 Java-Rätseln



Machen Sie Ihre GUIs hübsch und nützlich

Kathy Sierra & Bert Bates
Deutsche Übersetzung von Elke Buchholz und Lars Schulten

O'REILLY®

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

(trotz schwacher Beweise)

Und für Elke Buchholz, der wir diese deutsche
Ausgabe widmen.

# Die Köpfe hinter dieser Reihe

**Kathy** interessiert sich seit ihrer Zeit als Spiele-Designerin (sie hat Spiele für Virgin, MGM und Amblin' geschrieben) für Lerntheorie. Sie hat einen großen Teil des Formats der amerikanischen Originalreihe »Head First« entwickelt, aus der dieses Buch stammt, während sie im Rahmen des berufsbegleitenden Studienprogramms der UCLA im Studiengang Unterhaltung das Fach »New Media Authoring« unterrichtet hat. Später war sie Master Trainer bei Sun Microsystems und hat Java-Dozenten darin ausgebildet, wie man die neuesten Java-Techniken vermittelt, außerdem war sie leitende Entwicklerin der Java-Programmer- und Java-Developer-Zertifizierungsprüfungen von Sun. Gemeinsam mit Bert Bates hat sie die Konzepte von »Java von Kopf bis Fuß« aktiv im Unterricht für Hunderte von Trainern, Entwicklern und sogar Nicht-Programmierern eingesetzt. Zudem hat sie eine der größten Java-Community-Websites der Welt, *javaranch.com*, und das Blog »Creating Passionate Users« begründet.

Kathy war neben diesem Buch auch Koautorin von Head First Servlets & JSP, Head First EJB und Head First Design Patterns, das unter dem Titel »Entwurfsmuster von Kopf bis Fuß« auch auf Deutsch erschienen ist.

In ihrer Freizeit beschäftigt sie sich gern mit ihrem neuen Islandpferd, Skifahren, Laufen und der Lichtgeschwindigkeit.

*kathy@wickedlysmart.com*

**Bert** arbeitet als Software-Entwickler und -Architekt, aber die jahrzehntelange tägliche Arbeit mit künstlicher Intelligenz hat sein Interesse an Lerntheorie und computergestütztem Lernen gefördert. Seitdem bringt er Kunden das Programmieren bei. In letzter Zeit hat er im Entwicklerteam für einige von Suns Java-Zertifizierungsprüfungen mitgearbeitet.

Im ersten Jahrzehnt seiner Software-Karriere hat er die Welt bereist und für Rundfunk-Kunden wie Radio New Zealand, den Weather Channel und das Arts & Entertainment Network (A & E) gearbeitet. Eines seiner absoluten Lieblingsprojekte war der Aufbau einer kompletten Eisenbahnsystem-Simulation für Union Pacific Railroad.

Bert ist hoffnungslos dem *Go*-Spiel verfallen und arbeitet – schon etwas zu lange – an einem Go-Programm. Er spielt ganz gut Gitarre und versucht sich jetzt am Banjo. Zu seinen Lieblingsfreizeitvergnügen gehören Skifahren, Laufen und die Versuche, seinem Islandpferd Andi etwas beizubringen (oder von ihm zu lernen).

Bert ist Koautor der gleichen Bücher wie Kathy und bereits schwer mit dem nächsten Stoß Bücher beschäftigt.

Manchmal können Sie ihn auf dem IGS-Go-Server erwischen (unter dem Login-Namen jackStraw).

*terrapin@wickedlysmart.com*

Kathy und Bert versuchen, so viele E-Mails zu beantworten wie möglich, aber auf Grund des hohen Mail-Aufkommens und ihrer vielen Reiseverpflichtungen ist das schwierig. Der beste (schnellste) Weg, fachliche Hilfe zu diesem Buch zu erhalten, ist das *sehr* aktive Forum für Java-Anfänger auf javaranch.com.

# Über die Übersetzer dieses Buchs
## (die bei aller Berühmtheit nicht auf der Straße erkannt werden wollen)

**Lars Schulten** ist freier Übersetzer für IT-Fachliteratur und hat für den O'Reilly Verlag schon unzählige Bücher zu ungefähr allem übersetzt, was man mit Computern so anstellen kann. Eigentlich hat er mal Philosopie studiert, aber mit Computern schlägt er sich schon seit den Zeiten herum, in denen Windows laufen lernte. Die Liste der Dinge, mit denen er sich beschäftigt, ist ungefähr so lang, launenhaft und heterogen wie die seiner Lieblingsessen oder Lieblingsbücher.

Lars legt sich eben nicht gern fest. »Eine Klasse, eine Verantwortlichkeit», das ist sicher nicht seine Sache. Am besten funktioniert er, wenn man ihn als Universaladapter betrachtet und verdachtsweise einfach mal eine Methode aufruft und sich dann vom Ergebnis überraschen lässt.

Allein tritt er eigentlich nur auf, wenn er mal wieder versucht, den körperlichen Verfall mit sportlicher Betätigung aufzuhalten. Sonst ist er immer in Begleitung eines Buchs, seines Laptops oder Frederics unterwegs. Frederic ist vier Jahre alt und setzt gerne eine sehr kritische Miene auf, wenn Papa die Spielerei mit dem Conpuuta als Arbeit bezeichnet.

**Elke Buchholz** arbeitete als freiberufliche Übersetzerin und Lektorin. Sie hatte Naturwissenschaften studiert und anschließend einige Jahre lang für verschiedene Verlage Fachliteratur übersetzt, unter anderem auch Lehrbücher. Nach einer mehrjährigen Tätigkeit als Angestellte in einem Aachener Verlag war sie ab 2004 wieder selbstständig tätig.

Besonders gern übersetzte Elke Informatikfachliteratur, denn seit sie entdeckt hatte, wie spannend die Java-Programmierung sein kann, hat sie sich von diesem Fachgebiet nicht wieder losreißen können.

Passend zu ihrem Beruf und ihrem Lieblingshobby trug Elke BuchHolz einen »sprechenden Variablen(nach)-namen«. Als geborener Bücherwurm musste sie sich zwangsläufig auch beruflich irgendwann den Büchern zuwenden. Damit auch der zweite Teil ihres Namens nicht zu kurz kam, spielte sie in ihrer Freizeit begeistert ein Holzblasinstrument, die Klarinette.

Wenige Tage nach Drucklegung des Erstdrucks dieses Buchs starb Elke. Dieses Buch verdankt ihr sehr viel. Ihre Hingabe, ihr feiner Humor und ihre Klugheit machten die gemeinsame Arbeit an diesem Buch zu einer ganz besonderen Freude. Wir werden Dich nicht vergessen, Elke.

# Zur deutschen Übersetzung

Das leidige Thema der deutschen Umlaute im Code haben wir natürlich auch gründlich diskutiert. Der besseren Lesbarkeit halber haben wir schließlich in allen Arten von Bezeichnern Umlaute verwendet – auch in Klassennamen. Solange Sie innerhalb Ihres Systems bleiben und die Klassen nur dort kompilieren und ausführen, funktioniert dies in der Regel. Aber sobald Sie JARs erzeugen und/oder Ihre Programme auf anderen Plattformen einsetzen möchten, könnte es zu Problemen kommen. Sie sollten dann unbedingt in allen Verzeichnis- und Klassennamen (auch bei inneren Klassen!) auf Umlaute, auf das »ß» und Ähnliches verzichten. Profis, die meist sowieso zu englischen Bezeichnern neigen, empfinden Umlaute oft als irritierend und praxisfern – unser Anliegen ist es aber, dem lernenden Leser den Text, auch den Code, so leicht verdaulich zu präsentieren wie möglich.

# Der Inhalt (in der Übersicht)


| | | |
|---|---|---|
| | Einführung | xvii |
| 1 | Die Oberfläche durchbrechen: *eine Kostprobe* | 1 |
| 2 | Ein Ausflug nach Objekthausen: *ja, da gibt es Objekte* | 27 |
| 3 | Verstehen Sie Ihre Variablen: *elementare Datentypen und Referenztypen* | 49 |
| 4 | Wie sich Objekte verhalten: *Objektzustände beeinflussen Methodenverhalten* | 71 |
| 5 | Methoden mit Superkraft: *Schleifen, Operatoren und noch viel mehr* | 95 |
| 6 | Die Java-Bibliothek verwenden: *Sie müssen nicht alles selbst schreiben* | 125 |
| 7 | Besser leben in Objekthausen: *Zukunftsplanung* | 165 |
| 8 | Ernsthafte Polymorphie: *abstrakte Klassen und Interfaces nutzen* | 197 |
| 9 | Leben und Sterben eines Objekts: *Konstruktoren und Speicherverwaltung* | 235 |
| 10 | Zahlen, bitte! *Mathe, Zahlenformatierung und Statisches* | 273 |
| 11 | Riskantes Verhalten: *Exception-Handling* | 315 |
| 12 | Innen hui, außen GUI: *Grafische Oberflächen, Events und innere Klassen* | 353 |
| 13 | Mehr Schwung mit Swing: *Layout-Manager und Komponenten* | 399 |
| 14 | Speicherung von Objekten: *Serialisierung und E/A* | 429 |
| 15 | Eine Verbindung herstellen: *Netzwerk-Sockets und Multithreading* | 471 |
| 16 | Datenstrukturen: *Collections und Generics* | 529 |
| 17 | Veröffentlichen Sie Ihren Code: *Pakete, JARs und Deployment* | 581 |
| 18 | Verteilte Anwendungen: *RMI, Servlets, EJB und Jini* | 607 |
| A | Anhang A: *CodeKüchen-Finale* | 649 |
| B | Anhang B: *Die Top Ten der Themen, die es nicht ins Buch geschafft haben ...* | 659 |
| | Index | 677 |


# Der Inhalt (jetzt ausführlich)

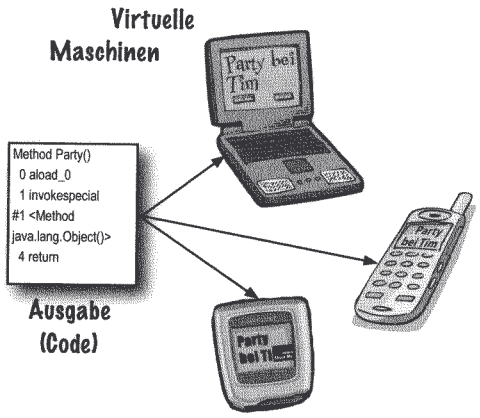
## Einführung

**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?


| | |
|---|---|
| Für wen ist dieses Buch? | xviii |
| Wir wissen, was Ihr Gehirn denkt | xix |
| Metakognition | xxi |
| Machen Sie sich Ihr Hirn untertan | xxiii |
| Was Sie für dieses Buch brauchen | xxiv |
| Die Fachgutachter | xxvi |
| Danksagungen | xxvii |

# 1 Die Oberfläche durchbrechen

**Java führt Sie an neue Orte.** Seit seinem bescheidenen Schritt an die Öffentlichkeit mit der (kümmerlichen) Version 1.02 hat Java mit seiner freundlichen Syntax, seinen objektorientierten Features, der Speicherverwaltung und natürlich mit dem Versprechen der Portabilität Programmierer 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!

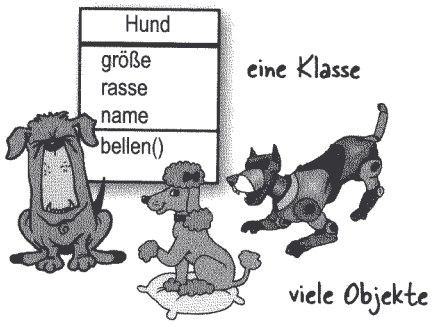




| | |
|---|---|
| Wie Java funktioniert | 2 |
| Code-Struktur in Java | 7 |
| Anatomie einer Klasse | 8 |
| Die main-Methode | 9 |
| Schleifen (*if*-Tests) | 11 |
| Bedingte Verzweigungen | 13 |
| »99 Bierflaschen« programmieren | 14 |
| Phras-O-Mat | 16 |
| Kamingespräche: Der Compiler und die JVM | 18 |
| Übungen und Rätsel | 20 |


# 2 Ein Ausflug nach Objekthausen

**Man hat mir gesagt, wir hätten es hier mit Objekten zu tun.** Aber in Kapitel 1 haben wir unseren gesamten Code in die main()-Methode gesteckt. Das ist nicht sonderlich objektorientiert! Jetzt müssen wir diese prozedurale Welt hinter uns lassen und damit beginnen, eigene Objekte zu machen. Wir werden uns ansehen, was die objektorientierte Entwicklung in Java zu einem solchen Vergnügen macht. Wir werden uns den Unterschied zwischen einer Klasse und einem Objekt ansehen. Wir werden uns ansehen, wie Objekte Ihr Leben verbessern.

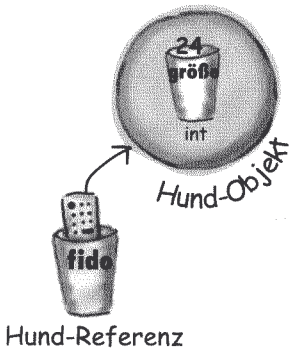




| | |
|---|---|
| Stuhlkriege (Stefan, der OO-Typ, gegen Harry, den prozeduralen Programmierer) | 28 |
| Vererbung (eine Einführung) | 31 |
| Methoden überschreiben (eine Einführung) | 32 |
| Was ist in einer Klasse? (Methoden, Instanzvariablen) | 34 |
| Ein erstes Objekt machen | 36 |
| main() verwenden | 38 |
| Der Code für das Ratespiel | 39 |
| Übungen und Rätsel | 42 |

# 3 Verstehen Sie Ihre Variablen

**Variablen gibt es in zwei Geschmacksrichtungen: elementare Typen und Referenztypen. Im Leben muss es doch mehr geben** als Integer, Strings und Arrays. Was ist, wenn Sie ein HaustierBesitzer-Objekt mit einer Hund-Instanzvariablen haben? Oder ein Auto mit einem Motor? In diesem Kapitel werden wir die Mysterien von Java-Typen offen legen und uns ansehen, was Sie als Variable *deklarieren* können, was Sie in eine Variable *stecken* können und was Sie mit einer Variablen *tun* können. Und schließlich werden wir verstehen, wie das Leben, das sich auf dem Garbage Collectible Heap abspielt, *wirklich* ist.

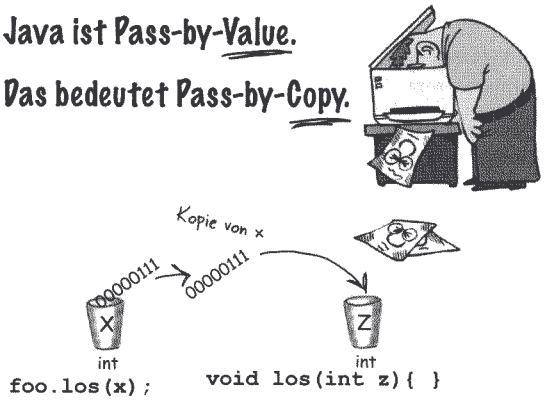


Hund-Referenz


| | |
|---|---|
| Eine Variable deklarieren (Java kümmert sich um *Typen*) | 50 |
| Elementare Typen (»Einen doppelten Espresso, bitte!«) | 51 |
| Java-Schlüsselwörter | 53 |
| Referenzvariablen (Fernsteuerung für ein Objekt) | 54 |
| Objekte deklarieren und zuweisen | 55 |
| Objekte auf dem Garbage Collectible Heap | 57 |
| Arrays (ein erster Blick) | 59 |
| Übungen und Rätsel | 63 |


# 4 Wie sich Objekte verhalten

**Zustände haben Auswirkungen auf Verhalten, und Verhalten hat Auswirkungen auf Zustände.** Wir wissen, dass Objekte **Zustände** und **Verhalten** haben, die von **Instanzvariablen** und **Methoden** repräsentiert werden. Nun sehen wir uns an, in welcher ***Beziehung*** Zustände und Verhalten zueinander ***stehen***. Objekte haben ein ***Verhalten***, das auf ihre ***Zustände*** wirkt. Anders gesagt, ***Methoden nutzen die Werte von Instanzvariablen*** – beispielsweise »wenn Hund weniger wiegt als 7 Kilo, mach Kläffgeräusch« oder »erhöhe Gewicht um 3 Kilo«. ***Ändern wir mal ein paar Zustände.***

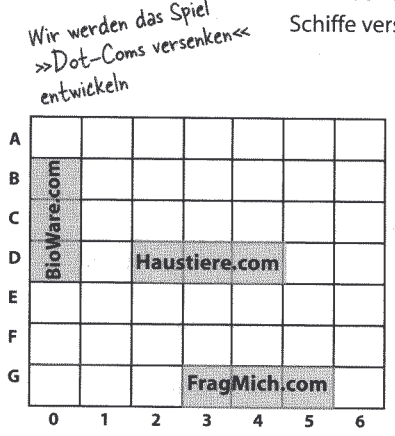




| | |
|---|---|
| Methoden nutzen Objektzustände (Die Größe beeinflusst das Bellen) | 73 |
| Methoden-Argumente und Rückgabetypen | 74 |
| Pass-by-Value (Die Variable wird *immer* kopiert) | 77 |
| Getter und Setter | 79 |
| Kapselung (tun Sie es oder machen Sie sich lächerlich) | 80 |
| Referenzen in Arrays verwenden | 83 |
| Übungen und Rätsel | 88 |

# 5 Methoden mit Superkraft

**Jetzt werden wir unseren Methoden etwas mehr Muskelkraft geben.** Wir haben uns an Variablen versucht, haben mit ein paar Objekten gespielt und haben etwas Code geschrieben. Jetzt brauchen wir mehr Werkzeuge. Wie ***Operatoren***. Und **Schleifen.** Es könnte auch nützlich sein, **Zufallszahlen zu generieren**. Oder **einen String in ein int umzuwandeln**, ja, das wäre cool. Und warum lernen wir nicht all den Kram, während wir was Echtes *schreiben*, um zu erfahren, wie es ist, wenn man ein Programm von der Pike auf schreiben (und testen) muss? **Vielleicht ein Spiel**, in dem wir Dot-Coms abschießen (wie bei Schiffe versenken).





| | |
|---|---|
| »Dot-Com versenken« entwickeln | 96 |
| EinfachesDotComSpiel (die simple Version) | 98 |
| Vorcode für EinfachesDotComSpiel schreiben | 100 |
| Testcode für EinfachesDotComSpiel schreiben | 102 |
| EinfachesDotComSpiel schreiben | 103 |
| Endgültigen Code für EinfachesDotComspiel schreiben | 106 |
| Zufallszahlen mit Math.random() generieren | 111 |
| Code-Fertiggericht für die Benutzereingabe in der Kommandozeile | 112 |
| *for*-Schleifen | 114 |
| Elementare Typen von großer Größe zu kleiner Größe casten | 117 |
| Einen String mit Integer.parseInt() in ein int umwandeln | 117 |
| Übungen und Rätsel | 118 |


# 6 Die Java-Bibliothek verwenden

**Java wird mit Hunderten vorgefertigter Klassen ausgeliefert.** Sie müssen das Rad nicht jedes Mal neu erfinden, wenn Sie wissen, wie Sie das, was Sie brauchen, in der Java-Bibliothek finden, die als die **Java-API** bezeichnet wird. *Schließlich haben Sie etwas Besseres zu tun.* Wenn Sie Code schreiben, können Sie sich ruhig auf die Teile beschränken, die bei Ihrer Anwendung besonders sind. Die Java-Kernbibliothek ist ein gigantischer Haufen von Klassen, die nur darauf warten, dass Sie sie als Bausteine einsetzen.

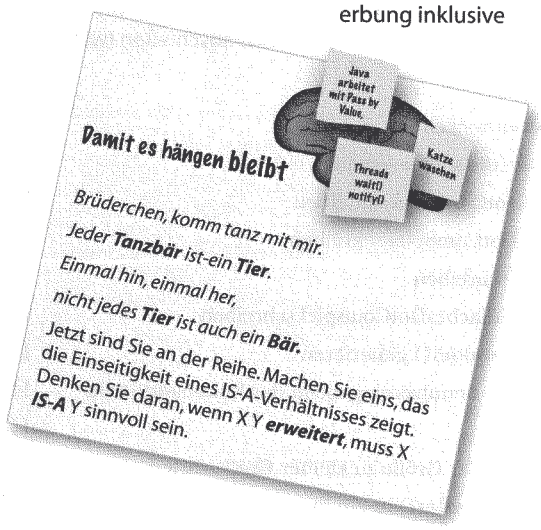
*»Gut zu wissen, dass es im Package java.util eine ArrayList gibt. Aber wie hätte **ich** das allein rausfinden können?«*

- Julia, 31, Hand-Model

| | |
|---|---|
| Den Fehler in unserem Dot-Com-Spiel analysieren | 126 |
| ArrayList (sich die Vorteile der Java API zunutze machen) | 132 |
| Den DotCom-Code reparieren | 138 |
| Das *richtige* Spiel bauen (»Dot-Com versenken«) | 140 |
| Vorcode für das *richtige* Spiel | 144 |
| Code für das *richtige* Spiel | 146 |
| Boolesche Ausdrücke | 151 |
| Die Bibliothek (die Java-API) verwenden | 154 |
| Packages verwenden (Import-Anweisungen und vollständige Namen) | 155 |
| Die HTML-API-Dokumentation und Referenzbücher nutzen | 158 |
| Übungen und Rätsel | 161 |

# 7 Besser leben in Objekthausen

**Planen Sie Ihre Programme mit Blick auf die Zukunft** Was wäre, wenn Sie Code schreiben könnten, den ein *anderer* **problemlos** erweitern kann? Und was wäre, wenn Ihr Code flexibel genug für diese verfluchten Spezifikationsänderungen in letzter Minute wäre? Wenn Sie auf den Polymorphie-Zug springen, lernen Sie die fünf Schritte zu besserem Klassen-Design, 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

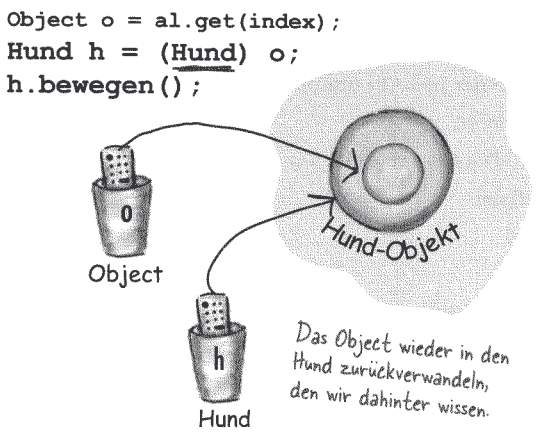




Vererbung verstehen (Unterklassen und Superklassen) 168
Einen Vererbungsbaum entwerfen (eine Tier-Simulation) 170
Codeverdopplung verhindern (mit Vererbung) 171
Methoden überschreiben 170
IST-EIN und HAT-EIN (die Badewannen-Frau) 177
Was kann man von der Superklasse erben? 180
Was *bringt* der ganze Vererbungskram? 182
Polymorphie (eine Supertyp-Referenz für Unterklassen-Objekt verwenden) 183
Regeln fürs Überschreiben (lassen Sie die Finger von den Argumenten und Rückgabetypen!) 190
Eine Methode überladen (nichts weiter als Methodennamen wiederverwenden) 191
Übungen und Rätsel 192


# 8 Ernsthafte Polymorphie

**Vererbung ist nur der Anfang.** Um Polymorphie richtig zu nutzen, brauchen wir Schnittstellen. 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 100% abstrakte Klasse. Was eine abstrakte Klasse ist? Das ist eine Klasse, die nicht instantiiert werden kann. Und wozu soll die gut sein? Lesen Sie das Kapitel...

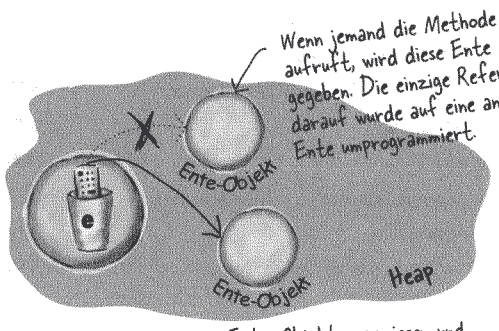




Manche Klassen sollten einfach *nicht* instantiiert werden 200
Abstrakte Klassen (*können nicht* instantiiert werden) 201
Abstrakte Methoden (müssen implementiert werden) 203
Polymorphie in Aktion 206
Die Klasse Object (die ultimative Superklasse für *alles*) 208
Objekte aus einer ArrayList nehmen (sie kommen als Typ Object raus) 211
Der Compiler prüft den Referenztyp (bevor er Sie eine Methode aufrufen lässt) 213
Finden Sie Ihr inneres Object 214
Polymorphe Referenzen 215
Eine Objektreferenz casten (im Vererbungsbaum nach unten verschieben) 216
»The Deadly Diamond of Death« (Problem der Mehrfachvererbung) 223
Interfaces verwenden (die beste Lösung) 224
Übungen und Rätsel 230

# 9 Konstruktoren und Garbage Collection

**Objekte werden geboren und Objekte sterben.** Sie herrschen über das Leben eines Objekts. Sie entscheiden, wie und wann ein Objekt *konstruiert* wird. Sie entscheiden, wann es aufgegeben werden soll. Der **Garbage Collector (gc)** will den Speicher wieder haben. In diesem Kapitel sehen wir uns an, wie Objekte erzeugt werden, wie sie leben, während sie leben, und wie Sie sie effektiv bewahren oder aufgeben. Das bedeutet, dass wir über den Heap, den Stack, Geltungsbereiche, Konstruktoren, Super-Konstruktoren, null-Referenzen und anderes reden werden.



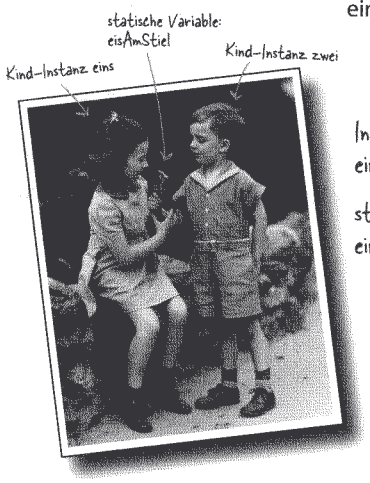

>>e<< erhält ein neues Ente-Objekt zugewiesen und gibt damit das ursprüngliche (erste) Ente-Objekt auf. Die erste Ente ist jetzt so gut wie tot.


| | |
|---|---|
| Der Stack und der Heap: wo das Leben spielt | 236 |
| Methoden auf dem Stack | 237 |
| Wo die *lokalen* Variablen leben | 238 |
| Wo die *Instanz*variablen leben | 239 |
| Das Wunder der Objekterzeugung | 240 |
| Konstruktoren (der Code, der ausgeführt wird, wenn Sie *new* sagen) | 241 |
| Den Zustand einer neuen Ente initialisieren | 243 |
| Überladene Konstruktoren | 247 |
| Superklassenkonstruktoren (Konstruktorverkettung) | 250 |
| Einen überladenen Konstruktor mit *this()* aufrufen | 256 |
| Das Leben eines Objekts | 258 |
| Garbage Collection (und Objekte auswählbar machen) | 260 |
| Übungen und Rätsel | 266 |


# 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 Trennzeichen unterteilt werden, damit man sie leichter lesen kann. Und wie steht es mit Datumsangaben? Als Erstes sehen wir uns einmal an, was es für eine Variable oder eine Methode bedeutet, statisch zu.

**Alle Instanzen der gleichen Klasse teilen sich ein einziges Exemplar der statischen Variablen.**



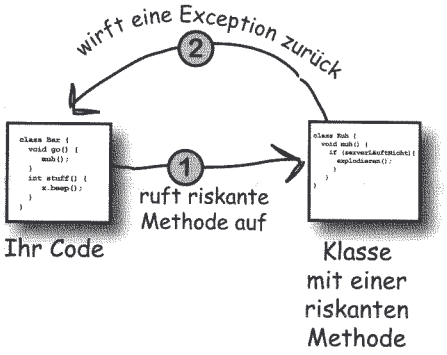

Instanzvariablen: eine pro Instanz

statische Variablen: eine pro Klasse


| | |
|---|---|
| Die Klasse Math (brauchen wir davon eine Instanz?) | 274 |
| Statische Methoden | 275 |
| Statische Variablen | 277 |
| Konstanten (statische finale Variablen) | 282 |
| Math-Methoden (random(), round(), abs() usw.) | 286 |
| Wrapperklassen (Integer, Boolean, Character usw.) | 287 |
| Autoboxing | 289 |
| Zahlen formatieren | 294 |
| Formatierung und Manipulation von Datumsangaben | 301 |
| Statische Importe | 307 |
| Übungen und Rätsel | 310 |

# 11 Riskantes Verhalten

**Manches passiert einfach. Die Datei ist nicht da. Der Server läuft nicht.** Und 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 all das Schlimme, das möglicherweise passiert, behandelt. Aber woher *wissen* Sie, wann eine Methode riskant ist? Und wo platzieren Sie den Code, der die ***Ausnahme***situation *behandelt*? In *diesem* Kapitel programmieren wir einen MIDI-Musikplayer, der die riskante Java-Sound-API benutzt – daher sollten wir das besser alles schnell rausfinden!





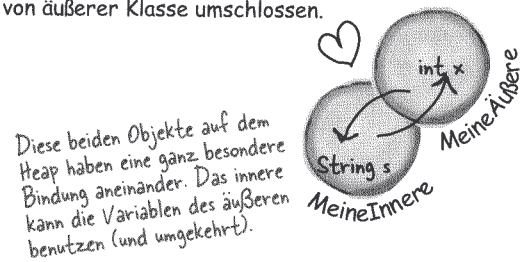
Bauen wir uns eine Musikmaschine 316
Wenn eine Methode, die Sie aufrufen wollen, riskant ist ... 319
Exceptions sagen: »Es ist etwas Schlimmes passiert.« 320
Der Compiler garantiert (er *prüft*), dass Sie das Risiko kennen 321
Das *try-and-catch*-Prinzip (Skateboarder) 322
Flusskontrolle in *try/catch*-Blöcken 326
Der finally-Block (»egal, was passiert, schalte den Herd aus!«) 327
Abfangen von Mehrfach-Exceptions 329
Eine Exception deklarieren (weichen Sie ihr einfach aus!) 335
Das Gesetz »Behandeln oder deklarieren« 337
CodeKüche (Sound erzeugen) 339
Übungen und Rätsel 348


# 12 Innen hui, außen GUI

**Sehen Sie den Tatsachen ins Auge: Um GUIs (grafische Benutzeroberflächen) kommen Sie nicht herum.** Selbst wenn Sie davon ausgehen, dass Sie den Rest Ihres Lebens serverseitigen Code schreiben, müssen Sie früher oder später Tools entwickeln, und dafür hätten Sie dann ganz sicher gern eine grafische Schnittstelle. Wir werden zwei Kapitel mit der Programmierung von GUIs zubringen und dabei grundlegende Features der Sprache Java kennen lernen, z.B. **Event-Handling** und **innere Klassen.** In diesem Kapitel setzen wir einen Button auf den Bildschirm und lassen ihn reagieren, wenn man ihn anklickt. Außerdem zeichnen wir etwas auf den Bildschirm, zeigen ein JPEG-Bild an und versuchen uns sogar an einer kleinen Animation.

```
class MeineÄußereKlasse {

  class MeineInnereKlasse {
    void los() {
    }
  }

}
```

Innere Klasse ist vollständig von äußerer Klasse umschlossen.

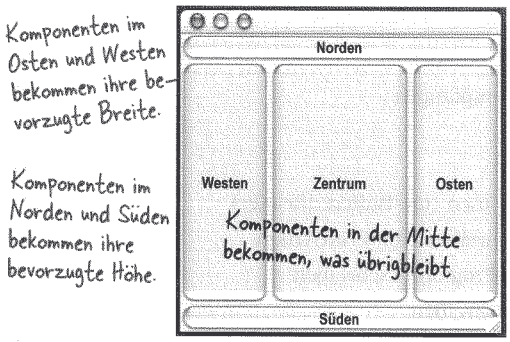




Ihre erste GUI 355
Wie man an ein Benutzerereignis kommt 357
Ein Listener-Interface implementieren 358
Wie man an das ActionEvent eines Buttons kommt 360
Dinge in eine GUI setzen 363
Schönes mit paintComponent() 365
Das Graphics2D-Objekt 366
Zwei oder mehr Widgets in einen Frame setzen 370
Innere Klassen kommen uns zu Hilfe 376
Animation 382
CodeKüche (Grafiken zum Beat der Musik einfügen) 386
Übungen und Rätsel 394

# 13 Mehr Schwung mit Swing

**Swing ist einfach.** Wenn es Ihnen nicht so wichtig ist, wo die Dinge auf dem Bildschirm landen. Swing-Code *sieht einfach aus*, aber dann kompilieren Sie das Ganze, führen es aus, sehen es sich an und denken: »Hey, *das* sollte aber nicht *dort* stehen ...« Der Grund dafür, dass der Code so leicht zu *schreiben* ist, ist der **Layoutmanager** – aber genau der macht es auch so *schwer kontrollierbar*. 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.

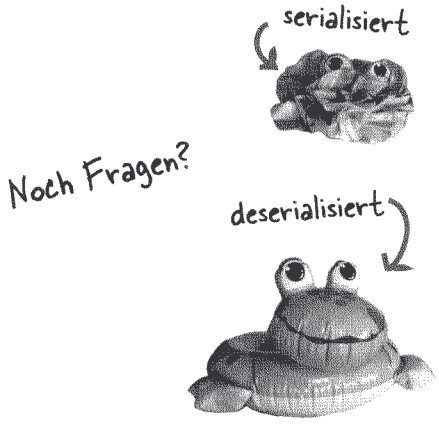




| | |
|---|---|
| Swing-Komponenten | 400 |
| Layoutmanager steuern Größe und Platzierung | 401 |
| Drei Layoutmanager: Border, Flow und Box | 403 |
| BorderLayout (managt fünf Bereiche) | 404 |
| FlowLayout (managt bevorzugte Größe und die Reihenfolge) | 408 |
| BoxLayout (ähnelt FlowLayout, kann Komponenten vertikal anordnen) | 411 |
| JTextField (für einzeilige Benutzereingaben) | 413 |
| JTextArea (für mehrzeilige Benutzereingaben, Scrollen) | 414 |
| JCheckBox (angeklickt?) | 416 |
| JList (scrollbare, auswählbare Liste) | 417 |
| CodeKüche (Ein Ungetüm – den BeatBox-Chat-Clienten erzeugen) | 418 |
| Übungen und Rätsel | 424 |


# 14 Speicherung von Objekten

**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 natürlich auf die mühsame Weise tun* – Sie befragen jedes Objekt und notieren dann akribisch den Wert jeder Instanzvariablen. Oder **Sie machen es auf die einfache, objektorientierte Weise** – indem Sie lediglich das Objekt selbst gefriertrocknen (serialisieren) und es mit Wasser anrühren (deserialisieren), wenn Sie es zurückhaben wollen.

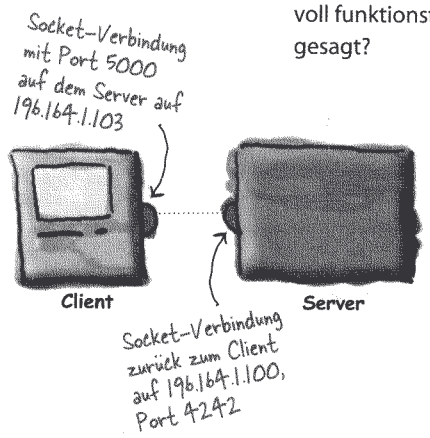




| | |
|---|---|
| Speicherung von Zuständen | 431 |
| Ein serialisiertes Objekt in eine Datei schreiben | 432 |
| Input- und Outputströme (Anschluss und Verkettung) | 433 |
| Serialisierte Objekte | 434 |
| Das Serializable-Interface implementieren | 437 |
| Transiente Variablen verwenden | 439 |
| Ein Objekt deserialisieren | 441 |
| In eine Textdatei schreiben | 447 |
| Die Klasse java.io.File | 452 |
| Aus einer Textdatei lesen | 454 |
| Einen String mit split() in mehrere Tokens spalten | 458 |
| CodeKüche | 462 |
| Übungen und Rätsel | 466 |

# 15 Eine Verbindung herstellen

**Nehmen Sie Verbindung mit der Außenwelt auf.** Das ist nicht schwer. Um all die Lowlevel-Netzwerk-Details kümmern sich Klassen in der java.net-Bibliothek. Einer der großen Vorteile von Java ist, dass das Senden und Empfangen über ein Netz einfache Eingabe/Ausgabe ist, lediglich mit einem etwas anderen Verbindungsstrom am Ende der Kette. In diesem Kapitel machen wir *Client*-Sockets. Wir machen *Server*-Sockets. Wir machen *Clients* und *Server*. Und wir lassen sie miteinander reden. Bevor Sie mit dem Kapitel durch sind, haben Sie einen voll funktionsfähigen und Multithread-tauglichen Chat-Client. Haben wir da gerade *Multithread* gesagt?

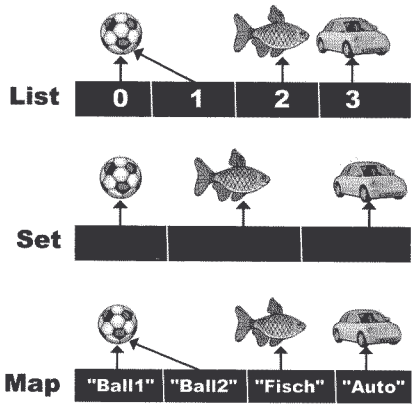




| | |
|---|---|
| Das Chat-Programm im Überblick | 473 |
| Verbinden, senden und empfangen | 474 |
| Netzwerk-Sockets | 475 |
| TCP-Ports | 476 |
| Daten von einem Socket lesen (mit BufferedReader) | 478 |
| Daten in einen Socket schreiben (mit PrintWriter) | 479 |
| Das TippDesTagesClient-Programm schreiben | 480 |
| Einen einfachen Server schreiben | 483 |
| Der TippDesTagesServer-Code | 484 |
| Einen Chat-Client schreiben | 486 |
| Mehrere Aufruf-Stacks | 490 |
| Einen neuen Thread starten | 492 |
| Das Runnable-Interface (Job des Threads) | 494 |
| Drei Zustände eines neuen Threads (neu, lauffähig, laufend) | 495 |
| Die lauffähig/laufend-Schleife | 496 |
| Der Thread-Scheduler (das ist seine Entscheidung, nicht Ihre!) | 497 |
| Einen Thread schlafen legen | 501 |
| Erzeugen und Starten von zwei Threads | 503 |
| Ehe in Gefahr: Ist dieses Paar noch zu retten? | 505 |
| Das Rainer-und-Monika-Problem in Codeform | 506 |
| Dinge sperren, um sie atomar zu machen | 510 |
| Jedes Objekt hat ein Schloss | 511 |
| Das gefürchtete »Problem des verlorenen Updates« | 512 |
| Synchronisierte Methoden (eine Sperre verwenden) | 514 |
| Thread-Deadlock | 516 |
| ChatClient mit mehreren Threads | 518 |
| Code-Fertiggericht für den EinfacherChatServer | 520 |
| Übungen und Rätsel | 524 |

# 16 Datenstrukturen

**Sortieren ist in Java ein Klacks.** Sie haben alle Werkzeuge zur Verfügung, um Daten zu sammeln und damit zu arbeiten, ohne Ihre eigenen Sortieralgorithmen schreiben zu müssen. Das Collections-Framework von Java enthält eine Datenstruktur für praktisch alles, was Sie jemals tun müssen. Sie wollen eine Liste, zu der Sie leicht etwas hinzufügen können? Sie wollen etwas mit Hilfe seines Namens suchen? Oder Sie brauchen eine Liste, die automatisch alles entfernt, was doppelt vorhanden ist? Sie möchten Ihre Kollegen danach sortieren, wie oft sie Ihnen in den Rücken gefallen sind? Es ist alles da ...

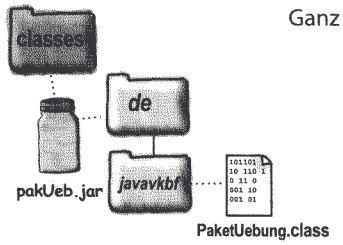


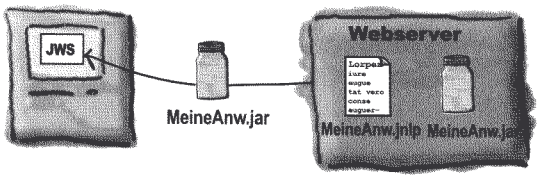


| | |
|---|---|
| Collections | 533 |
| Eine ArrayList mit Collections.sort() sortieren | 534 |
| Generics und Typsicherheit | 540 |
| Sachen sortieren, die das Comparable-Interface implementieren | 547 |
| Mit einem eigenen Comparator sortieren | 552 |
| Die Collection-API (List, Set und Map) | 557 |
| Verdopplungen mit HashSet verhindern | 559 |
| Überschreiben von hashCode() und equals() | 560 |
| HashMap | 567 |
| Wildcards für Polymorphie verwenden | 574 |
| Übungen und Rätsel | 576 |


# 17 Veröffentlichen Sie Ihren Code

**Es ist Zeit loszulassen.** Sie haben Ihren Code geschrieben. Sie haben ihn getestet. Sie haben ihn verbessert. Sie haben allen Bekannten erzählt, es wäre schön, wenn Sie nie wieder eine einzige Zeile Code sehen würden. Und doch haben Sie zu guter Letzt ein Kunstwerk geschaffen. Tatsächlich, das Ding läuft! Aber was nun? In den beiden letzten Kapiteln des Buchs befassen wir uns mit dem Organisieren, Packen und Deployment des Codes. Wir sehen uns lokale und entfernte Deployment-Möglichkeiten an, inklusive ausführbarer JARs, RMI und Servlets. Ganz ruhig – einige der coolsten Dinge, die Java bietet, sind einfacher, als Sie denken.



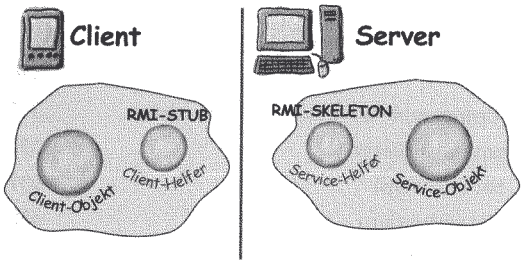





| | |
|---|---|
| Deployment-Optionen | 582 |
| Quellcode und class-Dateien getrennt halten | 584 |
| Machen Sie das JAR ausführbar | 585 |
| Das JAR ausführen | 586 |
| Stecken Sie Ihre Klassen in Pakete! | 587 |
| Pakete müssen eine passende Verzeichnisstruktur haben | 589 |
| Kompilieren und Ausführen mit Paketen | 590 |
| Kompilieren mit -d | 591 |
| Ein ausführbares JAR mit Paketen erzeugen | 592 |
| Java Web Start (JWS) fürs Deployment aus dem Web | 597 |
| So erstellt man eine JWS-Anwendung | 600 |
| Übungen und Rätsel | 601 |

# 18 Verteilte Anwendungen

**Entfernung muss kein Problem sein.** Sicher, es *ist* einfacher, wenn alle Teile Ihrer Anwendung an einem Ort sind, auf einem Heap, mit einer JVM, die über alles herrscht. Aber das ist nicht immer möglich. Und auch nicht immer wünschenswert. Was ist, wenn Ihr Programm leistungsintensive Berechnungen durchführen soll? Was ist, wenn Ihr Programm Daten aus einer sicheren Datenbank braucht? In diesem Kapitel lernen wir, wie man die erstaunlich einfache Java-Technik namens Remote Method Invocation (RMI) einsetzt. Wir werfen auch einen kurzen Blick auf Servlets, Enterprise Java Beans (EJB) und Jini.

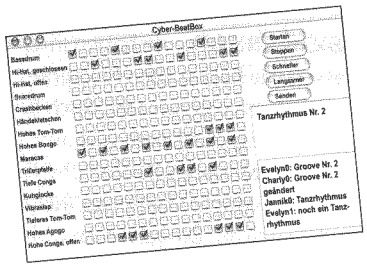




RMI zum Mitmachen, *sehr* ausführlich 614
Ein kurzer Blick auf Servlets 625
Ein *sehr* kurzer Blick auf Enterprise Java Beans (EJB) 631
Bezauberndes Jini 632
Den wirklich coolen Universaldienstbrowser erzeugen 636
Ende 648


# A Anhang A

**Das CodeKüchen-Finale!** Der vollständige Code für unser BeatBox-Client-Programm – Ihre Chance, ein Rockstar zu werden!





Der Client-Code 650
Der Server-Code 657


# B Anhang B

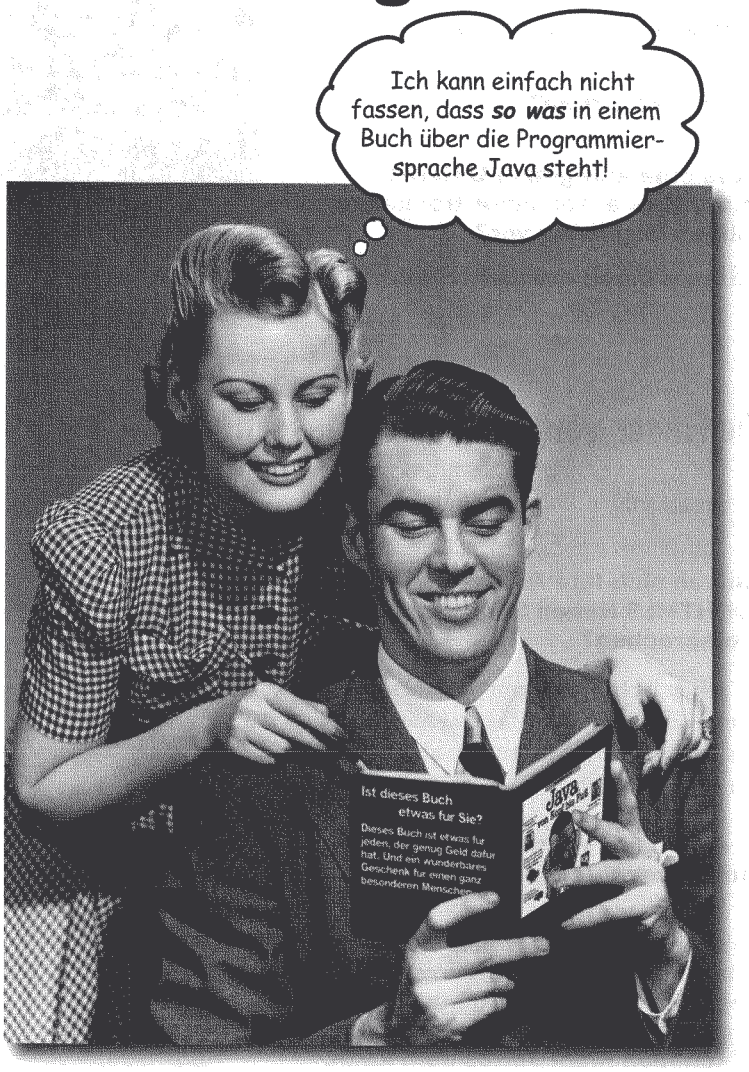
**Die Top Ten der Themen, die es nicht ins Buch geschafft haben.** Noch können wir Sie nicht ganz gehen lassen. Ein paar Sachen haben wir noch, aber dann sind Sie fertig. Diesmal meinen wir es ernst.


Die Top Ten 660


# I Index


677

# Einführung




In diesem Abschnitt beantworten wir die brennende Frage:
»Und? Warum STEHT so was 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 trockenen, langweiligen 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 mit Java-Fakten.**

# Wer sollte eher die Finger von diesem Buch lassen?

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

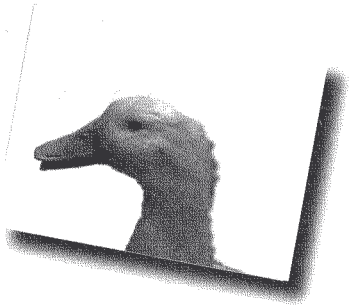
1. **Beschränken sich Ihre Programmierkenntnisse auf HTML? Haben Sie keine Erfahrung mit Skriptsprachen?**

   (Wenn Sie schon irgendetwas mit Schleifen oder Wenn-dann-Logik gemacht haben, werden Sie mit diesem Buch prima zurechtkommen, aber HTML-Tags allein reichen vielleicht nicht.)

2. **Sind Sie ein Top-C++-Entwickler, der ein Buch zum *Nachschlagen* sucht?**

3. **Haben Sie Angst, etwas Neues auszuprobieren? Ist Ihnen eine Zahnwurzelbehandlung lieber, als Streifen kombiniert mit Karos zu tragen? Glauben Sie, dass ein Technikfachbuch nicht seriös sein kann, wenn im Abschnitt über Speicherverwaltung das Bild einer Ente vorkommt?**

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



[Anmerkung der Marketing-Abteilung: Wer hat den Abschnitt darüber gestrichen, dass dieses Buch etwas für jeden Besitzer einer gültigen Kreditkarte ist? Und was ist mit dieser »Legen-Sie-Java-auf-den-Gabentisch«-Werbung? ... – Fred]

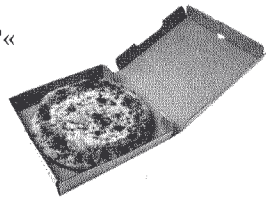
# Wir wissen, was Sie gerade denken.

»Kann das wirklich ein seriöses Programmierlehrbuch sein?«

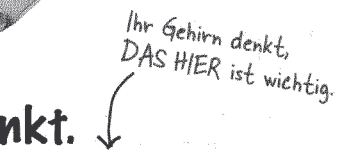
»Was ist mit all den Abbildungen?«

»Kann ich das auf diese Art wirklich lernen?«

»Riecht's hier nicht 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 etwas Ungewöhnliches. So ist es nun einmal gebaut, und es hilft Ihnen zu überleben.

Heutzutage ist es weniger wahrscheinlich, dass Sie von einem Tiger verputzt werden. Aber Ihr Gehirn hält immer noch Ausschau. Man weiß ja nie.

Also, was macht Ihr Gehirn mit all den gewöhnlichen, normalen Routinesachen, denen Sie begegnen? Es tut alles in seiner Macht stehende, damit es dadurch nicht bei seiner *eigentlichen* Arbeit gestört wird: Dinge zu erfassen, die wirklich *wichtig* sind. Es gibt sich nicht damit ab, die langweiligen Sachen zu speichern, sondern lässt diese gar nicht erst durch den »Dies-ist-offensichtlich-nicht-wichtig«-Filter.

Woher *weiß* Ihr Gehirn denn, was wichtig ist? Nehmen Sie an, Sie machen einen Tagesausflug und ein Tiger springt vor Ihnen aus dem Gebüsch: Was passiert dabei in Ihrem Kopf und Ihrem Körper?

Neuronen feuern. Gefühle werden angekurbelt. *Chemische Substanzen durchfluten Sie.*

Und so weiß Ihr Gehirn:

**Dies muss wichtig sein! Vergiss es nicht!**

Aber nun stellen Sie sich vor, Sie sind zu Hause oder in einer Bibliothek. In einer sicheren, warmen, tigerfreien Zone. Sie lernen. Bereiten sich auf eine Prüfung vor. Oder Sie versuchen, irgendein schwieriges Thema zu lernen, von dem Ihr Chef glaubt, Sie bräuchten dafür eine Woche oder höchstens zehn Tage.

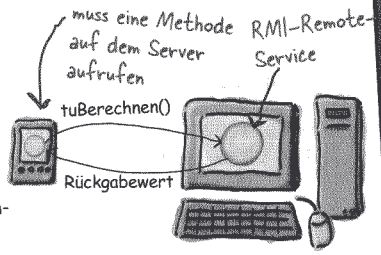
Da ist nur ein Problem: Ihr Gehirn versucht Ihnen einen großen Gefallen zu tun. Es versucht dafür zu sorgen, dass diese *offensichtlich* unwichtigen Inhalte nicht knappe Ressourcen verstopfen. Ressourcen, die besser dafür verwendet würden, die wirklich *wichtigen* Dinge zu speichern. Wie Tiger. Wie die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

Und es gibt keine einfache Möglichkeit, 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.«

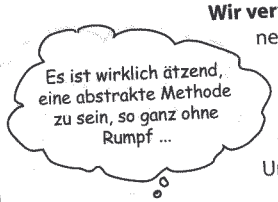
# Wir stellen uns unseren Leser als einen aktiv Lernenden vor.

**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.**

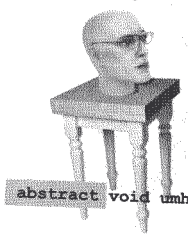
## Einige der Lernprinzipien dieser Buchreihe:



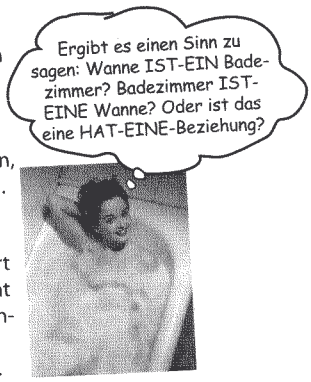

**Wir setzen Bilder ein.** An Bilder kann man sich viel besser erinnern als an Worte allein und lernt so viel effektiver (bis zu 89% Verbesserung bei Abrufbarkeits- und Lerntransferstudien). Außerdem werden die Dinge dadurch verständlicher. **Wir setzen Text in oder neben die Grafiken,** auf die sie 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 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?




**Wir wollen Sie dazu bringen, intensiver nachzudenken.** Mit anderen Worten: Falls Sie nicht aktiv Ihre Neuronen strapazieren, passiert in Ihrem Gehirn nicht viel. Ein Leser 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.




**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, was 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?

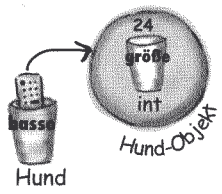
Da gibt es den langsamen, ermüdenden Weg oder den schnelleren, effektiveren Weg. Der langsame Weg führt über bloße Wiederholung. Natürlich ist Ihnen klar, dass Sie lernen und sich sogar an die langweiligsten Themen erinnern *können*, wenn Sie sich die gleiche Sache immer wieder einhämmern. Wenn Sie nur 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 voll passiver Zuhörer sitzen. Nicht nötig, wach zu bleiben.

Aber Abbildungen und ein lockerer Sprachstil sind erst der Anfang.

# Das haben WIR getan:

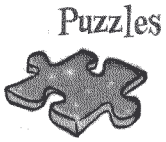



Wir haben ***Bilder*** verwendet, weil Ihr Gehirn auf visuelle Eindrücke eingestellt ist, nicht auf Text. Soweit es Ihr Gehirn betrifft, sagt ein Bild *wirklich* mehr als 1.024 Worte. Und dort, wo Text und Abbildungen zusammenwirken, haben wir den Text *in* die Bilder eingebettet, denn Ihr Gehirn arbeitet 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 ***Wiederholung*** eingesetzt, 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 ***unerwarter*** 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, den so ist es den meisten *Lesern* am liebsten.

Wir haben ***mehrere unterschiedliche Lernstile*** eingesetzt, denn vielleicht bevorzugen *Sie* ein Schritt-für-Schritt-Vorgehen, während jemand anders erst einmal den groben Zusammenhang verstehen und ein Dritter einfach nur ein Codebeispiel sehen möchte. Aber ganz abgesehen von den jeweiligen Lernvorlieben profitiert *jeder* davon, wenn er die gleichen Inhalte in unterschiedlicher Form präsentiert bekommt.

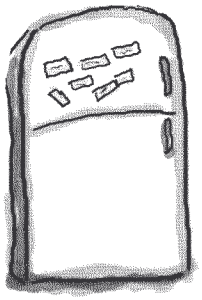
Wir liefern Inhalte für ***beide Seiten Ihres Gehirns,*** denn je mehr von Ihrem Gehirn Sie einsetzen, umso wahrscheinlicher werden Sie lernen und behalten und umso länger bleiben Sie konzentriert. Wenn Sie mit einer Seite des Gehirns arbeiten, bedeutet das häufig, dass sich die andere Seite des Gehirns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

Wir haben einen ***80/20***-Ansatz benutzt. 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 würden, wäre dass, als ob jemand anderes Ihr Training für Sie absolviert. 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 gleichen 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 wollen 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.

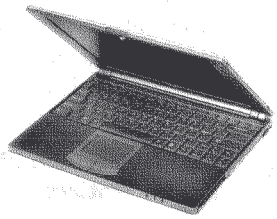
10. **Tippen Sie den Code ab und führen Sie ihn aus.**
    Tippen Sie die Codebeispiele ab und führen Sie sie aus. Dann können Sie herumexperimentieren und den Code verändern und verbessern (oder ihn kaputtmachen – das ist manchmal der beste Weg, um herauszufinden, was wirklich passiert). Bei langen Beispielen oder dem »Fertiggericht«-Code können Sie die Quellcode-Dateien von der Webseite zum Buch im Online-Katalog 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, abgekürzt IDE). Wir empfehlen Ihnen nachdrücklich, *ausschließlich* einen einfachen Texteditor zu verwenden, solange Sie dieses Buch noch nicht durchhaben (und vor allem, solange Sie Kapitel 16 noch nicht durchhaben). Eine IDE kann Ihnen einige der Details ersparen, die wirklich wichtig sind; daher ist es viel besser, wenn Sie das Ganze zunächst auf der Kommandozeile lernen. Erst wenn Sie das, was da passiert, wirklich verstehen, sollten Sie auf ein Tool umsteigen, das einen Teil des Prozesses automatisiert.

## INSTALLATION VON JAVA

- Sie benötigen ein **Java 2 Standard Edition SDK** (Software Development Kit) der Version **1.5** oder höher, falls Sie es nicht bereits haben. Für Linux, Windows und Solaris erhalten sie es kostenlos von java.sun.com (der Website von Sun für Java-Entwickler). Normalerweise braucht man nicht mehr als zwei Klicks, um von der Hauptseite auf die J2SE-Download-Seite zu gelangen. Holen Sie sich die neueste verfügbare *Nicht-Beta*-Version. Das SKD enthält alles, was Sie brauchen, um Java zu kompilieren und auszuführen.
  Wenn Sie mit Mac OS X 10.4 arbeiten, ist das Java SDK bereits installiert. Es ist Teil von OS X, daher brauchen Sie *nichts weiter* zu tun. Verwenden Sie eine ältere Version von OS X, haben Sie eine frühere Java-Version, die bei 95% des Codes in diesem Buch funktionieren wird.
  Hinweis: Dieses Buch basiert auf Java 1.5, aber aus merkwürdig unklaren Marketing-Gründen hat Sun dieses kurz vor der Veröffentlichung in Java 5 umbenannt, während als Versionsnummer für das SDK »1.5« beibehalten wurde. Wenn Sie also Java 1.5 lesen oder Java 5 oder Java 5.0 oder auch »Tiger« (der ursprüngliche Codename von Version 5) – *es bedeutet alles das Gleiche.* Java 3.0 und 4.0 hat es nie gegeben, sondern es gab einen Sprung von Version 1.4 zu 5.0, aber an manchen Stellen heißt es eben immer noch 1.5 statt 5. Fragen Sie nicht, warum. (Oh, und damit es noch lustiger wird: Sowohl Java 5 als auch Mac OS X 10.4 hatten als Codenamen »Tiger«! Und weil OS X 10.4 die Version von Mac OS ist, die man für die Ausführung von Java 5 benötigt, hört man die Leute manchmal auch von »Tiger auf Tiger« reden. Das bedeutet einfach: Java 5 auf OS X 10.4.)
- *Nicht* im SDK enthalten ist die **API-Dokumentation** – und die brauchen Sie! Gehen Sie noch einmal auf java.sun.com und holen Sie sich dort die J2SE-API-Dokumentation. Sie können auch online auf die API-Docs zugreifen, ohne sie herunterzuladen, aber das ist mühsam. Glauben Sie uns, es ist den Download wert.
- Sie brauchen einen **Texteditor.** Geeignet sind praktisch alle Texteditoren (vi, emacs, pico), auch die GUI-Editoren, die bei den meisten Betriebssystemen dabei sind. Notepad, Wordpad, Textedit usw. sind allesamt verwendbar, solange Sie sicherstellen, dass am Ende Ihres Quellcode-Dateinamens nicht ».txt« angehängt wird.
- Nach dem Herunterladen und Entpacken/Extrahieren/was auch immer (das ist versions- und betriebssystemabhängig) müssen Sie einen Eintrag zu Ihrer **PATH**-Umgebungsvariablen hinzufügen, der auf das Verzeichnis /bin innerhalb des Haupt-Java-Verzeichnisses verweist. Wenn beispielsweise das J2SDK ein Verzeichnis auf Ihrer Festplatte anlegt, das »j2sdk1.5.0« heißt, sehen Sie in diesem Verzeichnis nach; Sie finden dort das Verzeichnis bin, in dem die ausführbaren Java-Binärdateien (die Werkzeuge) liegen. Für dieses bin -Verzeichnis brauchen Sie einen PATH-Eintrag; dadurch weiß Ihr Terminal, wo es den Compiler *javac* findet, wenn Sie auf der Kommandozeile
  ```
  %javac
  ```
  eingeben.
  Hinweis: Wenn Sie Probleme bei der Installation haben, empfehlen wir Ihnen, das Forum für Java-Anfänger auf javaranch.com aufzusuchen! Aber das sollten Sie auf jeden Fall tun – egal ob Sie Probleme haben oder nicht.



Hinweis: Ein großer Teil des Codes aus diesem Buch kann von www.oreilly.de heruntergeladen werden.

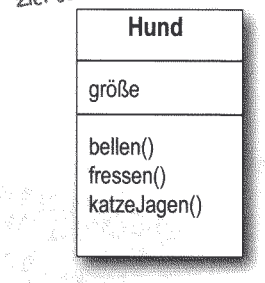
# Was Sie sonst noch wissen sollten:

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 anfangen, denn das Buch macht bestimmte Annahmen darüber, was Sie schon 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 benutzt, 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.




## Damit Sie es einfacher haben, schieben wir das Organisieren und Verpacken des Codes auf bis zum Ende des Buchs.

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, und wir werden sie daher eingehend behandeln. Aber wir schieben sie bis zum Ende des Buchs auf (Kapitel 17). So können Sie sich ganz entspannt an Java gewöhnen.

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

## Die Übungen am Kapitelende sind notwendig; die Rätsel sind optional. Die Antworten für beides stehen jeweils am Ende des Kapitels.

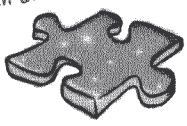
Eins müssen Sie über die »Puzzles« und »Rätsel« wissen – *es sind wirklich Rätsel.* So wie Knobeleien, Denksportaufgaben, Kreuzworträtsel usw. Die *Übungen* sind dazu gedacht, das Gelernte einzuüben, und sollten daher auf keinen Fall ausgelassen werden. Das gilt nicht für die Rätsel; manche davon stellen sogar eine ziemliche Herausforderung als Rätsel 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 können.

Aufgaben, die mit dem Trainingslogo (dem Laufschuh) markiert sind, sind Pflicht! Überspringen Sie sie nicht, wenn Sie es ernst meinen mit dem Java-Lernen.

## 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 sollen *Sie* im weiteren Verlauf Ihrer Lernerfahrung selbst entscheiden, ob *Ihre* Antworten richtig waren. (Einige Antwort*vorschläge* von uns finden Sie – in englischer Sprache auf wickedlysmart.com.)

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.

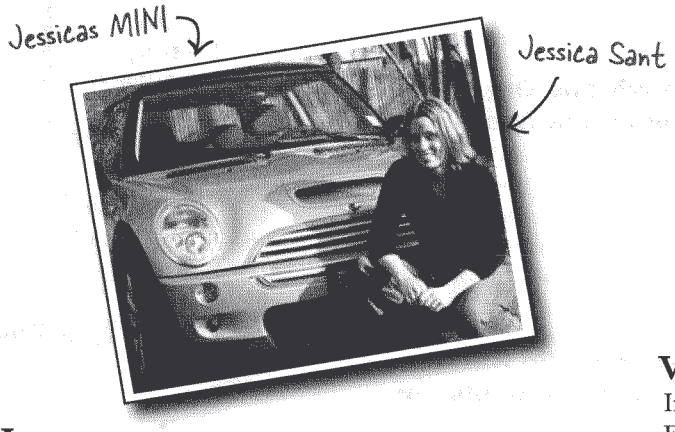


## Die Codebeispiele sind so kurz wie möglich.

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 beim 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.

# Fachgutachter

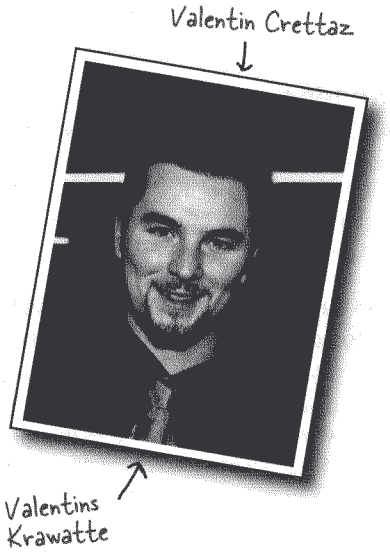
»Der Dank gebührt allen, aber Fehler liegen in der alleinigen Verantwortung des Autors ...« Glaubt das eigentlich irgend jemand? Sehen Sie die beiden Menschen auf dieser Seite? Falls Sie fachliche Fehler finden, dürfte es *deren* Schuld sein. :)




**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 fertig stellen 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 von Fachforen 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, Entwurfs- 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-Zertifikate. Zudem hatte er auch die Gelegenheit, als Koautor für den Whizlabs SCBCD Exam Simulator zu fungieren.

(Es hat uns einen Schock versetzt, ihn mit *Krawatte* zu sehen.)

Dank schulden wir

# ~~Schuld haben~~ außerdem:

***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 (bzw. 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 »Kopf-betonende« Cover der Reihe.

***Unseren tapferen Beta-Testern und Review-Team-Mitgliedern:***

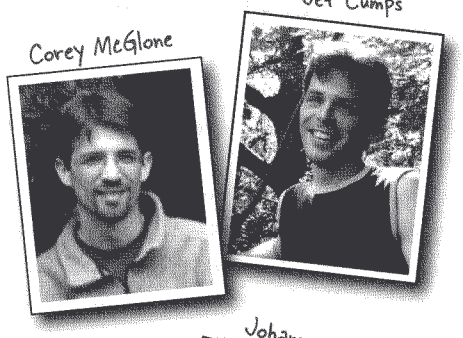
Ganz besonders danken wir dem Leiter unseres Javaranch-Review-Teams, **Johannes de Jong.** Du bist jetzt zum fünften Mal bei einem unserer »Head First«-Bücher dabei, und wir freuen uns mächtig, dass du trotzdem noch mit uns sprichst. **Jeff Cumps** ist zum dritten Mal dabei und findet unerbittlich die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

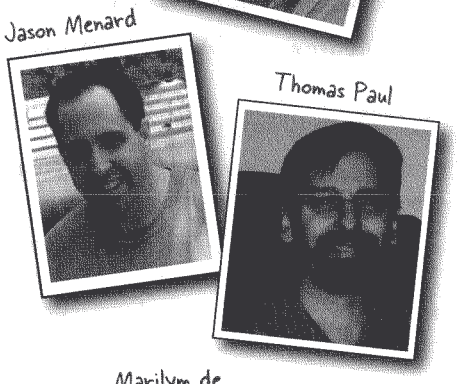
**Corey McGlone,** du bist Spitze. Und wir finden, du gibst auf javaranch.com die besten Erklärungen. Wahrscheinlich hast du gemerkt, dass wir eine oder zwei davon geklaut haben. **Jason Menard** hat uns mehr als einmal – und nicht nur bei Kleinigkeiten – in fachlicher Hinsicht gerettet, und **Thomas 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 Javaranch **Barry Gaunt** – eine große Hilfe bei der Neuauflage.

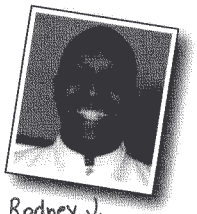
**Marilyn de Queiroz** hat uns bei *beiden* Auflagen des Buchs großartig unterstützt. Bei der ersten Auflage waren uns außerdem **Chris Jones, John Nyquist, James Cubeta, Terri Cubeta und Ira Becker** eine riesige Hilfe.

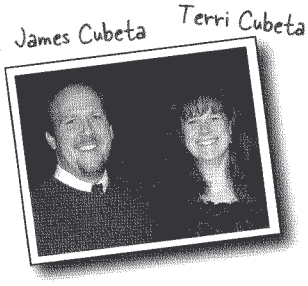
Besonderer Dank geht an einige der Head-First-Leute, die uns von Anfang an hilfreich zur Seite gestanden haben: **Angelo Celeste, Mikalai Zaikin** und **Thomas Duff** (*twduff.com*). Und danke an unseren wunderbaren Agenten, **David Rogelberg** von StudioB (aber mal ernsthaft, was ist mit den *Filmrechten*?).

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







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

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

Emiko Hori, Michael Taupitz, Mike Gallihugh, Manish Hatwalne, James Chegwidden, Shweta Mathnur, 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 aushelft.

***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, damit wir das hier pünktlich fertig stellen konnten.

**Sherry Dorris** – für die wirklich wichtigen Dinge.

***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).

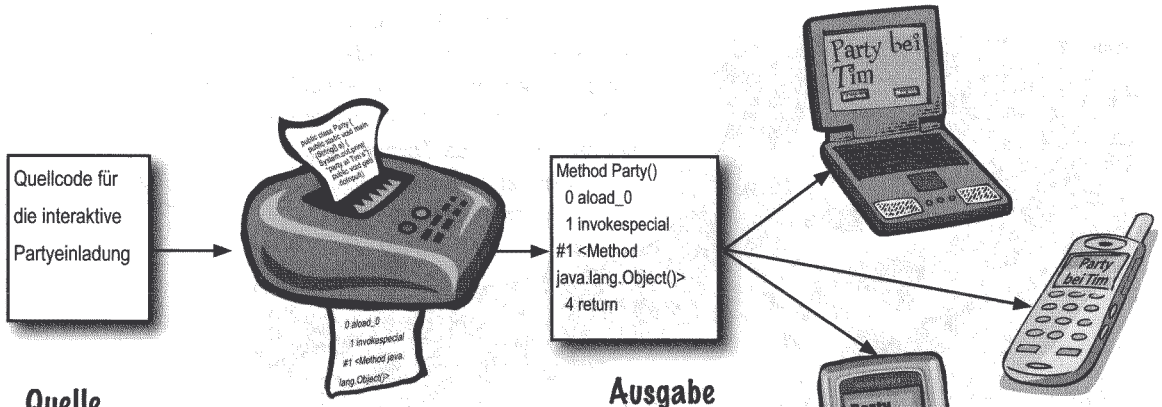
* Die große Zahl der Danksagungen hat folgenden Grund: Wir testen die Theorie, dass jeder, der in einem Buch bei den Danksagungen erwähnt wird, mindestens ein Exemplar davon kauft, wahrscheinlich sogar mehrere, wegen der Verwandtschaft und so. Wenn Sie bei den Danksagungen in unserem nächsten Buch erwähnt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

# Die Oberfläche durchbrechen

**Java führt Sie an neue Orte.** Seit seinem bescheidenen Schritt an die Öffentlichkeit mit der (kümmerlichen) Version 1.02 hat Java mit seiner freundlichen Syntax, seinen objektorientierten Features, der Speicherverwaltung und natürlich mit dem Versprechen der Portabilität Programmierer verführt. Die Verlockung des **Write-once/Run-anywhere** ist einfach zu groß. Die hingebungsvolle Gefolgschaft vermehrte sich explosionsartig, während Programmierer gegen Bugs, Beschränkungen und den Umstand kämpften, dass es schneckenlangsam war. Aber das ist Jahrhunderte 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 das trivialste Applet ans Laufen zu bringen. Aber *Sie*, *Sie* können sich von dem **geschmeidigeren, schnelleren, viel, viel mächtigeren** Java von heute tragen lassen.

# Wie Java funktioniert

**Das Ziel ist es, eine Anwendung zu schreiben (in diesem 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. Der Compiler prüft, ob Fehler vorhanden sind, und lässt Sie erst kompilieren, wenn er sicher ist, dass alles richtig läuft.

## Ausgabe (Code)

3

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

## Virtuelle Maschinen

4

Ihre Freunde haben keine physische Java-Maschine, aber alle haben eine ***virtuelle*** (in Software implementierte) Java-Maschine, die innerhalb ihrer elektronischen Spielzeuge läuft. Die virtuelle Maschine liest den Bytecode und führt ihn aus.

# Was Sie in Java machen

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

```
import java.awt.*;
import java.awt.event.*;
class Party {
  public void erstelleEinladung() {
    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 ihn als ***Party.java***.

```
Datei Bearbeiten Fenster Hilfe Flehen
%javac Party.java
```

## Compiler

**2**

Kompilieren Sie die Datei ***Party.java***, indem Sie `javac` (die Compiler-Anwendung) aufrufen. Wenn es zu keinen Fehlern kommt, erhalten Sie ein zweites Dokument namens ***Party.class***.

Die vom Compiler generierte Datei ***Party.class*** besteht aus *Bytecode*.

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

## Ausgabe (Code)

**3**

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

```
Datei Bearbeiten Fenster Hilfe Flehen
%java Party
```

## 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 zu Grunde liegende Plattform versteht, und führt Ihr Programm aus.

(Hinweis: Dies soll keine Anleitung sein ... wir beginnen gleich damit, richtigen Code zu schreiben, im Augenblick möchten wir Ihnen nur ein Gefühl dafür geben, wie das alles zusammenspielt.)

# Eine sehr kurze Java-Geschichte




## Java 1.0.2

250 Klassen

***Langsam***

Hübscher Name, hübsches Logo. Spaß beim Arbeiten. Viele Fehler. ***Applets*** sind die große Sache.

## Java 1.1

500 Klassen

***Etwas*** schneller

Mehr Fähigkeiten, ***freundlicher***. Beginnt, sehr ***beliebt*** zu werden. Besserer GUI-Code.

## Java 2 (Version 1.2 – 1.4)

2.300 Klassen

***Viel schneller***

Kann (manchmal) mit nativer Geschwindigkeit laufen. Ernsthaft, ***mächtig***. Kommt in drei Varianten: Micro Edition (J2ME), Standard Edition (J2SE) und Enterprise Edition (J2EE). Wird zur ***Sprache der Wahl*** für neue Enterprise-Anwendungen (insbesondere für webbasierte) und mobile Anwendungen.

## Java 5.0 (Version 1.5 und höher)

3.500 Klassen

***Mehr Power. Erleichtert die Entwicklung.***

Neben dem Hinzufügen von mehr als 1.000 zusätzlichen Klassen werden in Java 5.0 (auch bekannt als »Tiger«) größere Änderungen an der Sprache selbst vorgenommen, um (zumindest theoretisch) das Programmieren zu vereinfachen, und neue Features hinzugefügt, die in anderen Programmiersprachen beliebt waren.

## Spitzen Sie Ihren Bleistift

# Sehen Sie, wie einfach man Java schreiben kann.

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

```
int größe = 27;
String name = "Fido";
Hund meinHund = new Hund(name, größe);
x = größe - 5;
if (x < 15) meinHund.bellen(8);

while (x > 3) {
  meinHund.spielen();
}

int[] zahlListe = {2,4,6,8};
System.out.print("Hallo");
System.out.print("Hund: " + name);
String zahl = "8";
int z = Integer.parseInt(zahl);

try {
  leseDatei("meineDatei.txt");
}
catch(FileNotFoundException ex) {
  System.out.print("Datei nicht gefunden.");
}
```

Deklariert eine Integer-Variable namens »größe« und weist ihr den Wert 27 zu.

**F: Ich sehe hier Java 2 und Java 5.0. Gab es auch Java 3 und 4? Und warum heißt es Java 5.*0*, aber nicht 2.*0*?**

A: Die Freuden des Marketings ... als die Java-Version von 1.1 in 1.2 geändert wurde, waren die Änderungen an Java so dramatisch, dass das Marketing entschied, dass man einen ganz neuen »Namen« bräuchte. Also nannten sie es ***Java 2***, obwohl die eigentliche Version von Java 1.2 war. Aber die Versionen 1.3 und 1.4 wurden immer noch als ***Java 2*** betrachtet. Es gab nie ein Java 3 oder 4. Bei der Java-Version 1.5 entschied das Marketing erneut, dass die Änderungen so dramatisch seien, dass ein neuer Name her müsste (und die meisten Entwickler stimmten zu). Also haben sie sich die möglichen Optionen vorgenommen. Die nächste Zahl in der Namenssequenz wäre »3« gewesen. Aber es schien ziemlich verwirrend, Java 1.5 ***Java 3*** zu nennen. Also entschlossen sie sich, es ***Java 5.0*** zu nennen, analog zur »5« in Version »1.5«.

Das ursprüngliche Java, die Versionen 1.02 (die erste offizielle Veröffentlichung) bis 1.1, war also einfach nur »Java«. Die Versionen 1.2, 1.3 und 1.4 waren »Java 2«. Und seit Version 1.5 wird Java »Java 5.0« genannt. Aber manchmal nennt man es auch nur »Java 5« (ohne das ».0«) oder »Tiger« (der ursprüngliche Codename). Wir haben keine Ahnung, was bei der *nächsten* Version passieren wird ...

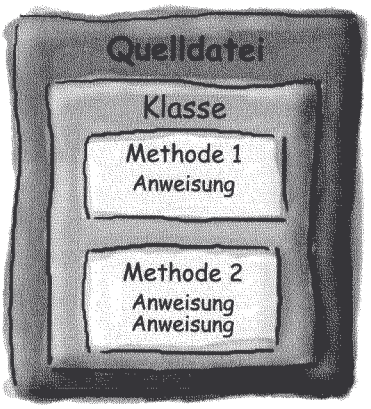
## Spitzen Sie Ihren Bleistift Antworten

# Sehen Sie, wie einfach man Java schreiben kann.

***Machen Sie sich keine Sorgen, wenn Sie jetzt noch nichts davon 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 in der Vergangenheit gearbeitet haben, wird manches davon ganz einfach sein. Wenn nicht, kümmern Sie sich nicht darum. *Wir werden das schon schaffen ...*

| | |
|---|---|
| `int größe = 27;` | Deklariert eine Integer-Variable namens »größe« und weist ihr den Wert 27 zu. |
| `String name = "Fido";` | Deklariert einen String namens »name« und weist ihm den Wert »Fido« zu. |
| `Hund meinHund = new Hund(name, größe);` | Deklariert eine neue Hund-Variable »meinHund« und bewirkt, dass der neue Hund »name« und »größe« verwendet. |
| `x = größe - 5;` | Zieht von 27 (dem Wert von »größe«) 5 ab und weist den Wert einer Variablen namens »x« zu. |
| `if (x < 15) meinHund.bellen(8);` | Wenn x (mit dem aktuellen Wert 22) kleiner als 15 ist, wird dem Hund gesagt, dass er 8-mal bellen soll. |
| | |
| `while (x > 3) {` | Die Schleife wird so lange durchlaufen, wie x größer als 3 ist ... |
| `  meinHund.spielen();` | Sagt dem Hund, 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[] zahlListe = {2,4,6,8};` | Deklariert eine Liste von Integer-Variablen namens »zahlListe« und packt 2,4,6 und 8 in die Liste. |
| `System.out.print("Hallo");` | Gibt »Hallo« aus ... wahrscheinlich auf der Kommandozeile. |
| `System.out.print("Hund: " + name);` | Gibt auf der Kommandozeile »Hund: Fido« aus (der Wert von »name« ist »Fido«). |
| `String zahl = "8";` | Deklariert einen String »zahl« und weist ihm den Wert »8« zu. |
| `int z = Integer.parseInt(zahl);` | 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 ... |
| `  leseDatei("meineDatei.txt");` | Liest eine Textdatei namens »meineDatei.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, nicht geklappt hat ... |
| `  System.out.print("Datei nicht gefunden.");` | Wenn das, was wir versucht haben, misslungen ist, wird auf der Kommandozeile »Datei nicht gefunden« ausgegeben. |
| `}` | Sieht aus, als wäre alles, was in { } steht, das, was gemacht werden soll, wenn das »Versuchen« nicht funktioniert hat ... |

# Code-Struktur 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 Erweiterung *.java*) enthält eine ***Klassendefinition***. Die Klasse stellt einen *Teil* Ihres Programms dar – auch wenn eine ganz winzige Anwendung vielleicht nur aus einer Klasse besteht. Die Klasse muss in einem Paar geschweifter Klammern stecken.

```
public class Hund {

}
```
Klasse

## Was kommt in eine Klasse?

Eine Klasse hat eine oder mehrere ***Methoden***. Die ***bellen***-Methode der Klasse Hund sollte Anweisungen dazu enthalten, wie der Hund bellen soll. Ihre Methoden müssen *innerhalb* einer Klasse (d.h. innerhalb der geschweiften Klammern der Klasse) deklariert werden.

```
public class Hund {
   void bellen() {
   }
}
```
Methode

## Was kommt in eine Methode?

In die geschweiften Klammern einer Methode schreiben Sie die Anweisungen dazu, wie die Methode ausgeführt werden soll. Methoden-*Code* ist im Wesentlichen ein Satz von Anweisungen, und für den Augenblick können Sie sich eine Methode als so etwas wie eine Funktion oder eine Prozedur vorstellen.

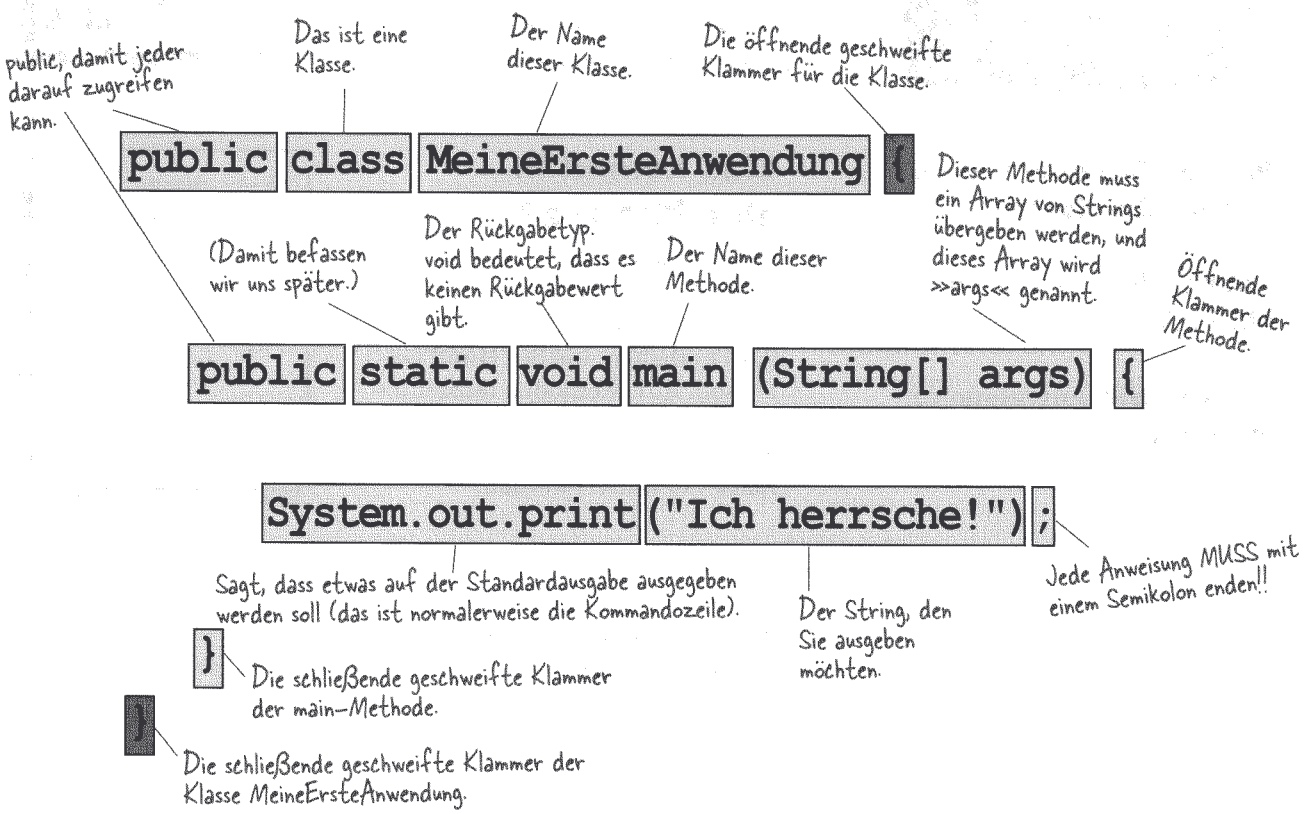
```
public class Hund {
   void bellen() {
     Anweisung1;
     Anweisung2;
   }
}
```
Anweisungen

# Anatomie einer Klasse

Wenn die JVM gestartet wird, sucht sie nach der Klasse, die Sie auf der Kommandozeile angegeben haben. Dann beginnt sie, nach einer besonders geschriebenen Methode zu suchen, die genau so aussieht:

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

Als Nächstes führt die JVM alles zwischen den geschweiften Klammern { } Ihrer main()-Methode aus. Jede Java-Anwendung muss mindestens eine **Klasse** und mindestens eine **main**-Methode haben (nicht eine main-Methode pro *Klasse*, lediglich eine pro *Anwendung*).




Das müssen Sie sich jetzt nicht alles gleich in den Kopf hämmern ...

Dieses Kapitel soll Sie nur auf den Weg bringen.

# Eine Klasse mit einer main-Methode schreiben

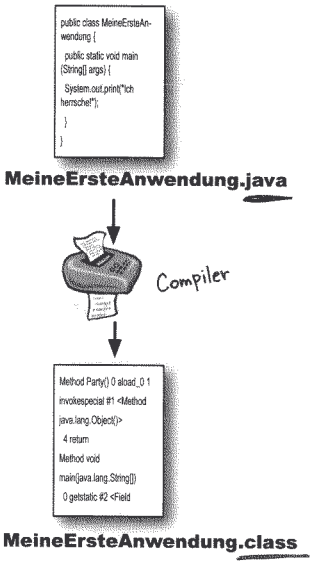
In Java kommt alles in jeweils eine **Klasse**. Sie geben Ihre Quellcode-Datei (mit der Erweiterung *.java*) ein und kompilieren sie dann zu einer neuen Klassendatei (mit der Erweiterung *.class*). Wenn Sie Ihr Programm ausführen, führen Sie eigentlich eine *Klasse* aus.

Ein Programm auszuführen bedeutet, der Java Virtual Machine (JVM) Folgendes zu sagen: »Lade die Klasse **`Hallo`** und führe dann ihre **`main()`**-Methode aus. Führe die Ausführung fort, bis der ganze Code in main fertig ist.«

In Kapitel 2 werden wir uns diese ganze *Klassengeschichte* gründlicher ansehen, aber im Augenblick müssen Sie nur wissen, ***wie Sie Java-Code 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.

Egal wie umfangreich Ihr Programm ist (anders gesagt, egal wie viele *Klassen* Ihr Programm verwendet) – es muss eine **main**()-Methode geben, damit der Stein ins Rollen kommt.




```
public class MeineErsteAnwendung {

  public static void main (String[] args) {
    System.out.println("Ich herrsche!");
    System.out.println("über die Welt");
  }

}
```

**1 Speichern**

`MeineErsteAnwendung.java`

**2 Kompilieren**

```
javac MeineErsteAnwen-
dung.java
```

**3 Ausführen**

```
Datei Bearbeiten Fenster Hilfe Kreischen
% java MeineErsteAnwendung
Ich herrsche!
über die Welt
```

## Was kann man in der main-Methode sagen?

Wenn man einmal in main (oder einer *beliebigen* anderen Methode) ist, beginnt das Vergnügen. Sie können all die üblichen Sachen sagen, die Sie in den meisten Programmiersprachen sagen können, ***um den Computer dazu zu bringen, etwas zu machen.***

Beispielsweise kann Ihr Code der JVM sagen:

### 1 Mach was

**Anweisungen:** Deklarationen, Zuweisungen, Methodenaufrufe usw.

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

### 2 Mach was wieder und wieder

**Schleifen:** *for* und *while*

```
while (x > 12) {
   x = x - 1;
}
for (int x = 0; x < 10; x = x + 1) {
   System.out.print("x ist jetzt " + x);
}
```

### 3 Mach was, falls ...

**Verzweigungen:** *if/else*-Tests

```
if (x == 10) {
   System.out.print("x ist 10 ");
} else {
   System.out.print("ist nicht 10");
}
if ((x < 3) & (name.equals("Dirk"))) {
   System.out.println("Vorsicht");
}
System.out.print("Diese Zeile wird auf alle Fälle
ausgeführt.");
```

### Syntax-Kram

★ Jede Anweisung muss mit einem Semikolon enden.

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

★ Ein Einzeilenkommentar beginnt mit zwei Schrägstrichen.

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

★ Whitespace ist in der Regel egal.

```
x     =     3 ;
```

★ Variablen werden mit einem **Namen** und einem **Typ** deklariert (in Kapitel 3 werden Sie alles über die Java-Typen erfahren).

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

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

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

```
while (mehrBälle == true) {
   weiterJonglieren( ) ;
}
```

## Schleifen, Schleifen, Schleifen ...

Java besitzt drei Standard-Schleifenkonstrukte: *while, do-while* und *for*. Die volle Schleifenladung kommt später im Buch. Aber das dauert noch etwas. Deswegen machen wir jetzt erst einmal *while*.

Die Syntax (von der Logik ganz zu schweigen) ist so einfach, dass Sie jetzt sicher schon eingeschlafen sind. Alles, was im Schleifen*block* steht, wird so lange ausgeführt, wie irgendeine Bedingung wahr ist. Der Schleifenblock wird von einem Paar geschweifter Klammern eingeschlossen. Alles, was wiederholt werden soll, muss also in diesem Block stehen.

Der Schlüssel für eine Schleife ist die *Testbedingung*. In Java ist eine Testbedingung ein Ausdruck, der zu einem *Booleschen Wert* ausgewertet wird – also etwas, das entweder ***true*** (wahr) oder ***false*** (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. Aber sagen Sie so was wie »SoLange *Bob* mache Eiskugeln«, haben Sie keinen echten Test. Damit das funktioniert, müssen Sie es in etwas wie »SoLange Bob schnarcht ...« oder »SoLange Bob *nicht* trägtUmhang ...« ändern.

## 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 4 zu
while (x > 3) {
  // der Schleifencode wird ausgeführt, weil
  // x größer als 3 ist
  x = x - 1; // oder wir würden für immer
             // in der Schleife hängen
}
int z = 27; //
while (z == 17) {
  // der Schleifencode wird nicht
  // ausgeführt, weil z nicht gleich 17 ist
}
```

## Es gibt keine Dummen Fragen

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

A: Java ist eine objektorientierte Sprache (eine OO-Sprache). Das hat nichts mit den guten alten Zeiten dampfgetriebener Compiler zu tun, in denen man eine monolithische Quelldatei mit einem Haufen von Prozeduren geschrieben hatte. In Kapitel 2 werden Sie lernen, dass eine Klasse eine Blaupause für ein Objekt ist und dass in Java fast alles ein Objekt ist.

**F: Muss ich in jede Klasse, die ich schreibe, eine main-Methode packen?**

A: Nein. Ein Java-Programm kann Dutzende (sogar Hunderte) von Klassen verwenden, von denen trotzdem vielleicht nur eine *einzige* eine main-Methode hat – die, die die Ausführung des Programms beginnt. Sie können aber auch Testklassen schreiben, die main-Methoden haben, mit denen Ihre *anderen* Klassen getestet werden.

**F: In meiner anderen Sprache kann ich Boolesche Tests auch auf Integer-Werten durchführen. Könnte ich in Java so was auch sagen:**

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

A: Nein. Die Typen für *Boolesche Werte* und *Integer-Werte* sind in Java nicht kompatibel. Da das Ergebnis einer Testbedingung ein Boolescher Wert sein *muss*, können Sie nur Variablen direkt (ohne die Verwendung eines Vergleichsoperators) testen, deren Typ ***boolean*** (d.h. ein Boolescher Wert) ist. Beispielsweise können Sie sagen:

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

## Beispiel für eine `while`-Schleife:

```
public class Schleifchen {
  public static void main (String[] args) {
    int x = 1;
    System.out.println("Vor der Schleife");
    while (x < 4) {
      System.out.println("In der Schleife");
      System.out.println("Der Wert von x ist " + x);
        x = x + 1;
    }
    System.out.println("Nach der Schleife");
  }
}
```

```
% java Schleifchen
Vor der Schleife
In der Schleife
Der Wert von x ist 1
In der Schleife
Der Wert von x ist 2
In der Schleife
Der Wert von x ist 3
Nach der Schleife
```

Das ist die Ausgabe.

### Punkt für Punkt

- Anweisungen enden mit einem Semikolon ;
- Codeblöcke werden durch ein Paar geschweifter Klammern definiert { }
- Eine *int*-Variable mit einem Namen und einem Typ deklarieren: **int x;**
- Der **Zuweisungsoperator** ist *ein* Gleichheitszeichen **=**
- Der **Gleichheitsoperator** besteht aus *zwei* Gleichheitszeichen **==**
- Eine *while*-Schleife führt alles innerhalb ihres Blocks (der durch geschweifte Klammern definiert wird) so lange aus, wie die *Testbedingung* ***true*** ist.
- Wenn die Testbedingung ***false*** ist, wird der Codeblock der *while*-Schleife nicht ausgeführt. Die Ausführung geht unmittelbar mit dem Code weiter, der *nach* dem Schleifenblock folgt.
- Eine Testbedingung wird in runde Klammern gesetzt: `while (x == 4) { }`

# Bedingte Verzweigungen

In Java ist ein *Wenn*-Test (*if*-Test) im Wesentlichen mit einem Booleschen Test in einer *while*-Schleife identisch – aber statt »***SoLange*** es Bier gibt ...« sagen Sie »***Wenn*** es Bier gibt ...«.

```
class IfTest {
  public static void main (String[] args) {
    int x = 3;
    if (x == 3) {
      System.out.println("x ist 3");
    }
    System.out.println("Das wird immer ausgeführt");
  }
}
```

```
% java IfTest
x ist 3
Das wird immer ausgeführt
```

Codeausgabe

Der Code oben führt die Zeile, die »x ist 3« ausgibt, nur aus, wenn die Bedingung (*x* ist gleich 3) wahr ist. Unabhängig davon, ob der Test wahr ist, wird die Zeile ausgeführt, die »Das wird immer ausgeführt« ausgibt. In Abhängigkeit vom Wert von *x* machen also entweder ein oder zwei Anweisungen eine Ausgabe.

Aber wir können der Bedingung ein *else* (sonst) hinzufügen, damit wir auch so was sagen können: »*Wenn* es Bier gibt, schreib das Programm weiter, *sonst* geh Bier kaufen und fahre dann fort mit...«

```
class IfTest2 {
  public static void main (String[] args) {
    int x = 2;
    if (x == 3) {
      System.out.println("x ist 3");
    } else {
      System.out.println("x ist NICHT 3");
    }
    System.out.println("Das wird immer ausgeführt");
  }
}
```

```
% java IfTest2
x ist NICHT 3
Das wird immer ausgeführt
```

Neue Ausgabe

## System.out.print() vs. System.out.println()

Wenn Sie aufgepasst haben (und natürlich haben Sie das), ist Ihnen sicher aufgefallen, dass wir zwischen **print()** und **println()** gewechselt haben.

**Ist Ihnen der Unterschied klar geworden?**

System.out.***println()*** fügt einen Zeilenumbruch (newline) ein (denken Sie sich println*()* als **printnewline**), System.out.***print()*** hingegen fährt fort, auf die *gleiche* Zeile auszugeben. Wenn alles auf einer eigenen Zeile ausgegeben werden soll, verwenden Sie println. Wenn alles in einer Zeile zusammenstehen soll, verwenden Sie print.

## Spitzen Sie Ihren Bleistift

***Bei folgender Ausgabe:***

```
% java DooBee
DooBeeDooBeeDo
```

***Tragen Sie den fehlenden Code ein:***

```
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");
  }
 }
}
```

# Eine ernsthafte Geschäftsanwendung schreiben

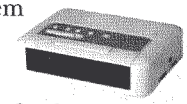
Setzen wir all Ihre neuen Java-Fähigkeiten mal für etwas Praktisches ein. Wir brauchen eine Klasse mit einer *main()*-Methode, eine *int*- und eine *String*-Variable, eine *while*-Schleife und einen *if*-Test. Dann das Ganze noch etwas aufpolieren, und Sie haben das Backend für diese Geschäftsanwendung im Handumdrehen zusammengebaut. Aber *bevor* Sie sich den Code auf dieser Seite ansehen, denken Sie einen Augenblick darüber nach, wie Sie den alten Kinderklassiker »99 Bierflaschen« programmieren würden.

```
public class BierLied {
  public static void main (String[] args) {
    int bierAnzahl = 99;
    String wort = "Flaschen";

    while (bierAnzahl > 0) {

      if (bierAnzahl == 1) {
        wort = "Flasche"; // Singular, wie in EINE Flasche
      }

      System.out.println(bierAnzahl + " " + wort + " Bier im Kühlschrank");
      System.out.println(bierAnzahl + " " + wort + " Bier.");
      System.out.println("Hol eine raus.");
      System.out.println("Und lass sie rumgehen.");
      bierAnzahl = bierAnzahl - 1;

      if (bierAnzahl > 0) {
        System.out.println(bierAnzahl + " " + wort + " Bier im Kühlschrank");
      } else {
        System.out.println("Kein Bier mehr im Kühlschrank");
      } // Ende else
    } // Ende while-Schleife
  } // Ende main-Methode
} // Ende Klasse
```

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.

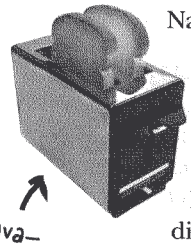
# Montagmorgen bei Bob

Montagmorgen klingelt der Wecker bei Bob wie an jedem anderen Wochentag um 8:30. Aber Bob hat ein wildes Wochenende hinter sich und tastet nach dem SCHLUMMER-Knopf. Das ist der Augenblick, in dem die Action losgeht und die Java-gesteuerten Geräte zum Leben erwachen.




Zuerst sendet der Wecker der Kaffeemaschine eine Nachricht:* »Hey, der Geek pennt wieder ein, warte mit dem Kaffee noch 12 Minuten.«

Die Kaffeemaschine schickt dem Motorola™-Toaster die Nachricht: »Warte mit dem Toast, Bob schnarcht noch.«




Der Wecker wiederum schickt Bobs Nokia Navigator™-Handy diese Nachricht: »Ruf Bobs 9-Uhr-Termin an und sag ihm, dass wir uns etwas verspäten.«

Und schließlich schickt der Wecker Sams (Sam ist der Hund) WiFi-Halsband das so vertraute Signal, das »Hol die Zeitung, aber mach dir keine Hoffnung auf einen Spaziergang!« sagt.

Ein paar Minuten später klingelt der Wecker wieder. Und *wieder* haut Bob auf SCHLUMMER und setzt damit das Geplapper seiner Geräte in Gang. Dann klingelt der Wecker ein drittes Mal. Aber während Bob noch nach dem Schlummer-Knopf sucht, sendet der Wecker das »Spring auf und bell«-Signal an Sams Halsband. Geschockt und jetzt hellwach steht Bob auf, dankbar, dass seine Java-Fähigkeiten und ein kleiner Ausflug zu Conrad Electronic™ seinen Alltag vereinfacht haben.

***Sein Toast ist getoastet.***

***Sein Kaffee dampft.***

***Seine Zeitung wartet auf ihn.***

Wieder ein neuer wunderbarer Morgen in ***Das Java-gesteuerte Haus.***

***Auch Sie können sich ein Java-gesteuertes Haus einrichten.*** Halten Sie sich an vernünftige Lösungen mit Java, Ethernet und der Jini-Technologie. Halten Sie sich von so genannten »Plug and Play«-Lösungen (was eigentlich heißt: »Steck's zusammen und spiel dann damit für die nächsten drei Tage, damit es irgendwann funktioniert!«) oder »portablen« Plattformen fern. Bobs Schwester Betty hat eine von diesen *anderen* ausprobiert, und die Ergebnisse waren, wie sagt man doch, nicht sonderlich ansprechend oder sicher. Ziemlich schade, das mit ihrem Hund ...

Könnte diese Geschichte wahr sein? Ja und nein. Es *gibt* zwar Java-Versionen, die auf Geräten wie PDAs, Handys (*insbesondere* Handys), Pagern, Smart Cards oder anderem laufen – aber es könnte schwierig werden, einen Toaster oder ein Hundehalsband zu finden. Doch selbst wenn Sie keine Java-fähige Version Ihres Lieblingsspielzeugs finden, können Sie es immer noch ausführen, als *wäre* es ein Java-Gerät, indem Sie es über eine andere Schnittstelle (beispielsweise Ihren Laptop) steuern, auf der Java *läuft*. Das wird als Jini-*Surrogatarchitektur* bezeichnet. Ja, Sie *können* dieses Geek-Traumhaus haben.

*Per *IP Multicast*, falls Sie sich für das Protokoll interessieren.

Na gut, das Bier-Lied war keine *wirklich* ernsthafte Geschäftsanwendung. Brauchen Sie immer noch etwas Praktisches, das Sie Ihrem Boss vorzeigen können? Sehen Sie sich 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 Ding zwischen den Anführungszeichen) eingeben. Der Code lässt sich dann nicht mehr kompilieren. Die Bindestriche, die Sie auf dieser Seite sehen, sind echt und dürfen eingegeben werden. Aber Enter dürfen Sie nur drücken, NACHDEM Sie einen String geschlossen haben.

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

1  // drei Wortgruppen erstellen, aus denen ausgewählt wird
   String[] wortListeEins = {"verlässliche",
"erfolgsorientierte", "webbasierte", "allumfassende",
"clevere", "kundenorientierte", "pfadkritische",
"dynamische", "konkurrenzfähige", "verteilte",
"zielgerichtete"};

   String[] wortListeZwei = {"gepowerte ", "haftende ",
"Mehrwert-", "zentrierte ", "geclusterte ",
"proaktive ","Out-of-the-Box-", "positionierte ",
"vernetzte ", "fokussierte ", "kraftvolle ", "geordnete ",
"geteilte ", "kooperative ", "beschleunigte ",
"Multi-Tier-","Enterprise-","B2B-","Frontend-"};

   String[] wortListeDrei = {"Schicht", "Endstufe",
"Lösung", "Architektur", "Kernkompetenz", "Strategie",
"Kooperation", "Ausrichtung", "Räumlichkeit", "Vision",
"Dimension", "Mission"};

2  // ermitteln, wie viele Wörter sich in jeder Liste befinden
   int einsLänge = wortListeEins.length;
   int zweiLänge = wortListeZwei.length;
   int dreiLänge = wortListeDrei.length;

3  // drei Zufallszahlen generieren, um Wörter aus den Listen
   herauszuziehen
   int rand1 = (int) (Math.random() * einsLänge);
   int rand2 = (int) (Math.random() * zweiLänge);
   int rand3 = (int) (Math.random() * dreiLänge);

4  // eine Phrase zusammenbauen
   String phrase = wortListeEins[rand1] + " " +
   wortListeZwei[rand2] + wortListeDrei[rand3];

5  // eine Phrase ausgeben
   System.out.println("Was wir brauchen, ist eine " +
   phrase);
   }
}
```

# 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 noch nicht *genau* verstehen, was in allen Zeilen passiert. Meine Güte, Sie haben schließlich noch das ganze Buch vor sich. Entspannen Sie sich einfach. Das ist nur ein kurzer Blick aus einer erfolgsorientierten kraftvollen geclusterten B2B-Vision.

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[] haustiere = {"Fido", "Zeus", "Dino"};
```

Die einzelnen Wörter stehen in Anführungszeichen (wie es alle braven Strings müssen) 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 in einer Liste 14 Wörter sind, müssen wir eine Zufallszahl zwischen 0 und 13 erzeugen (Java-Arrays sind nullbasiert, das erste Element befindet sich also an der Position 0, das zweite an Position 1 und das letzte bei einem Array mit 14 Elementen an Position 13). Praktischerweise ist es für ein Java-Array ein Vergnügen, Ihnen seine Länge mitzuteilen. Sie müssen es einfach nur fragen. Beim Array haustiere würden wir Folgendes sagen:

```
int x = haustiere.length;
```

`x` würde dann den Wert 3 enthalten.

3. Wir brauchen drei Zufallszahlen. Java bietet ganz Out-of-the-Box, vakuumverpackt und kernkompetent einen Satz von mathematischen Methoden (die Sie sich im Augenblick ruhig als Funktionen vorstellen können). Die Methode `random()` liefert eine Zufallszahl zwischen 0 und nicht ganz 1 zurück. Wir müssen sie also mit der Anzahl der Elemente aus der Liste (der Array-Länge) multiplizieren, die wir verwenden. Dann müssen wir das Ergebnis zu einer Ganzzahl zurechtbiegen (Dezimalzahlen sind nicht erlaubt), also fügen wir einen Cast (eine Typumwandlung, Details dazu erhalten Sie in Kapitel 4) ein. Das ist das Gleiche, als hätten wir eine Fließkommazahl, die wir in eine Ganzzahl umwandeln wollen:

```
int x = (int) 24.6;
```

4. Jetzt bauen wir die Phrase auf, indem wir ein Wort aus jeder der drei Listen auswählen und sie zusammenpappen (und dabei zwischen den Wörtern Leerzeichen einfügen). Wir verwenden den »+«-Operator, der die String-Objekte verknüpft oder *konkateniert* (eigentlich ziehen wir den technischeren Begriff »zusammenpappen« vor). Um ein Element aus einem Array abzurufen, geben Sie dem Array die Indexnummer (Position) des Elements an, das Sie haben möchten:

```
String s = haustiere[0]; // s ist jetzt der String "Fido"
s = s + " " + "ist ein Hund"; // s ist jetzt "Fido ist ein Hund"
```

5. Schließlich geben wir die Phrase auf der Kommandozeile aus ... et voilà! *Wir sind im Marketing.*

**Was wir hier brauchen, ist eine ...**

allumfassende zielgerichtete Strategie

dynamische Out-of-the-Box-Endstufe

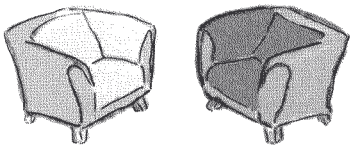
clever vernetzte Kernkompetenz

erfolgsorientierte gepowerte Kooperation

verteilte B2B-Vision

verlässliche geclusterte Kundenorientierung

## Kamingespräche



Heute Abend: **Der Compiler und die JVM streiten sich 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 ans *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 doch gleich von Hand Bytecode schreiben und mir geben. Dauert nicht mehr lange und Sie sind arbeitslos, Freund.

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

**Der Compiler**

Dieser Ton gefällt mir überhaupt nicht.

Entschuldigung, aber ohne *mich* – was würden Sie da ausführen? Es gibt schon einen *Grund,* dass Java so entworfen wurde, dass es einen Bytecode-Compiler verwendet, falls es Sie interessiert. Wenn Java eine rein interpretierte Sprache wäre, bei der die virtuelle Maschine – zur Laufzeit – den Quellcode frisch aus dem Texteditor übersetzen müsste, würde ein Java-Programm mit Schneckengeschwindigkeit ablaufen. Java hatte es schon schwer genug, die Anwender zu überzeugen, dass es endlich schnell und leistungsfähig genug für die meisten Aufgaben ist.

Entschuldigung! Aber das ist eine ziemlich ignorante (um nicht zu sagen arrogante) Sichtweise. Es *ist* zwar – *theoretisch* – richtig, dass Sie dazu in der Lage sind, jeden ordentlich formatierten Bytecode auszuführen, auch wenn der nicht aus einem Java-Compiler kommt, aber in der Praxis ist das absurd. Von Hand Bytecode schreiben? Das ist, als würden Sie Textverarbeitung betreiben, indem Sie rohes PostScript schreiben. Und ich würde es vorziehen, Sie würden darauf *verzichten,* mich »Freund« zu nennen.

## 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*? Ohohoh. 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.

Darauf kannst du dich verlassen. *Freund.*

## Der Compiler

Denken Sie daran, dass Java eine streng typisierte Sprache ist. Das heißt, dass ich nicht zulassen darf, dass Variablen Daten des falschen Typs 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 eine 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 Datentyp-Verletzungen, die ich eben beschrieben habe, könnten in einem Programm Chaos verursachen, wenn zugelassen würde, dass sie auftreten. 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 Leute 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: Wenn ich nicht so annähernd 99% der möglichen Probleme verhindern würde, würden Sie sich ständig festfahren. Aber jetzt ist anscheinend unsere Zeit abgelaufen. Wir müssen das also in einem späteren Gespräche noch mal ausdiskutieren.

# Code-Magneten

Ein funktionierendes Java-Programm ist vollkommen durcheinander geraten. 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 hinzu, wie Sie benötigen!

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

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

```
class MischMasch1 {
  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 MischMasch1
a-b c-d
```

# Spielen Sie Compiler

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

### A

```
class Übung1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      if ( x > 3) {
        System.out.println("großes x");
      }
    }
  }
}
```

### B

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

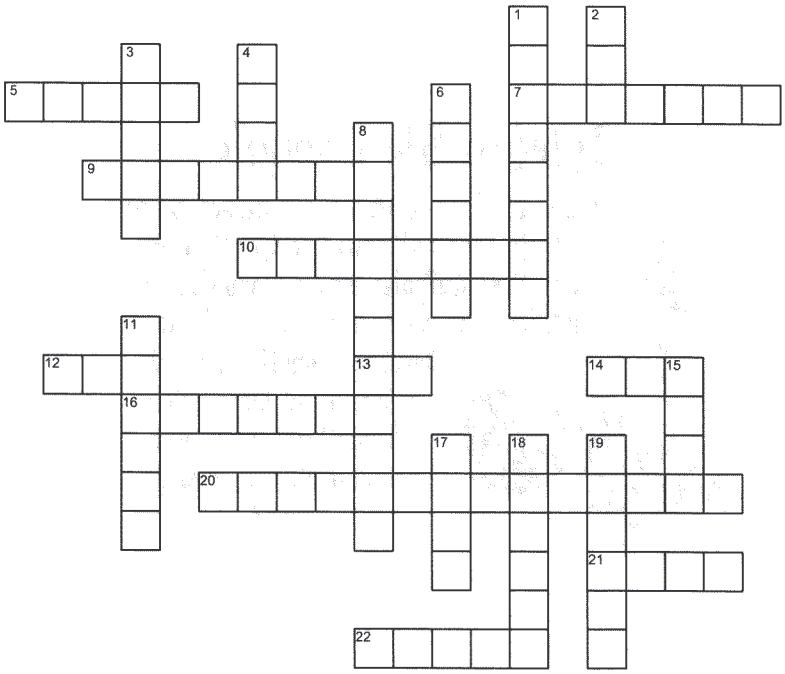
### C

```
class Übung1b {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("kleines x");
      }
    }
}
```

# JavaKreuz 7.0

Geben wir Ihrer rechten Gehirnhälfte mal etwas Arbeit.

Das 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 Hightech-Welt eingestreut.



## Waagerecht

5 Solange die Bedingung zufrieden ist.

7 Hiermit tun die Dinge etwas.

9 Kann man nicht festnageln.

10 Manchmal ist einmal nicht genug!

12 Akronym für Prozessor.

13 Die Abteilung, ohne die nichts geht.

14 Bytecode-Schlucker.

16 Wofür eine Eingabeaufforderung gut ist.

20 Sagt etwas.

21 Unzuverlässigste Laptop-Komponente.

22 Kann nicht in beide Richtungen gehen.

## Senkrecht

1 Quellcode-Schlucker.

2 Variablentyp für Zahlen.

3 Kein Integer.

4 Kommandozeilen-Aufrufer.

6 Schockierender Modifizierer.

8 Eine neue Klasse oder Methode ankündigen.

11 Offenes Haus.

15 Muss man nur einmal haben.

17 Kommt mit leeren Händen zurück.

18 Eine ordentliche Mannschaft von Zeichen.

19 »Dinge«-Halter.

## Vermischte Nachrichten

Unten sehen Sie ein kurzes Java-Programm. Ein Block des Programms fehlt. Ihre Herausforderung besteht darin, **die möglichen Codeblöcke** (auf der linken Seite) **den Ausgaben zuzuordnen**, die Sie sehen würden, wenn der Block eingefügt würde. Nicht alle Zeilen mit Ausgabe werden verwendet, und einige Zeilen mit Ausgaben können mehrfach verwendet werden. Ziehen Sie Linien, die die Codeblock-Kandidaten mit den passenden Kommandozeilen-Ausgaben verbinden. (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 kommen die Codeblock-Kandidaten rein.
      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
```

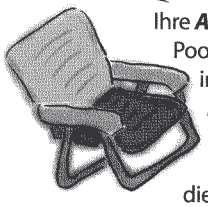
```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

# Pool-Puzzle



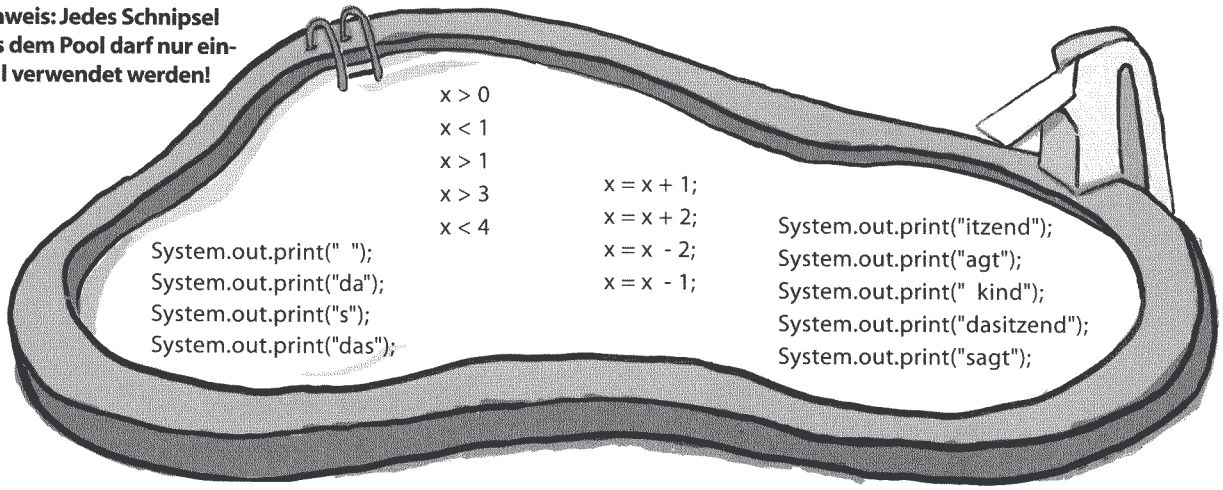
Ihre ***Aufgabe*** ist es, die Codeschnipsel aus dem Pool zu nehmen und sie in die leeren Zeilen im Code einzusetzen. Jedes Schnipsel darf ***nur einmal*** verwendet werden, aber Sie werden nicht alle Schnipsel benötigen. Ihr ***Ziel*** ist es, eine Klasse zu erstellen, die sich kompilieren lässt und die hier gezeigte Ausgabe erzeugt. Lassen Sie sich nicht täuschen – das ist schwerer, als es aussieht.

**Ausgabe**

```
Datei Bearbeiten Fenster Hilfe Schummeln
%java PoolPuzzleEins
da sagt
dasitzend
das kind
```

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

    while ( _____ ) {

      _____
      if ( x < 1 ) {
        _____
      }
      _____

      if ( _____ ) {

        _____

        _____
      }
      if ( x == 1 ) {

        _____
      }
      if ( _____ ) {

        _____
      }
      System.out.println("");

      _____
    }
  }
}
```

**Hinweis: Jedes Schnipsel aus dem Pool darf nur einmal verwendet werden!**



# Lösungen zu den Übungen

## Code-Magneten:

```
class MischMasch1 {
  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 Dichter
% java MischMasch1
a-b c-d
```

**A**

```
class Übung1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      x = x + 1;
      if ( x > 3) {
        System.out.println("großes x");
      }
    }
  }
}
```

Dieses Programm wird kompiliert und läuft (ohne Ausgabe). Aber wenn ihm nicht die eine Zeile hinzugefügt wird, hängt es auf ewig in einer endlosen »while«-Schleife!

---

**B**

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

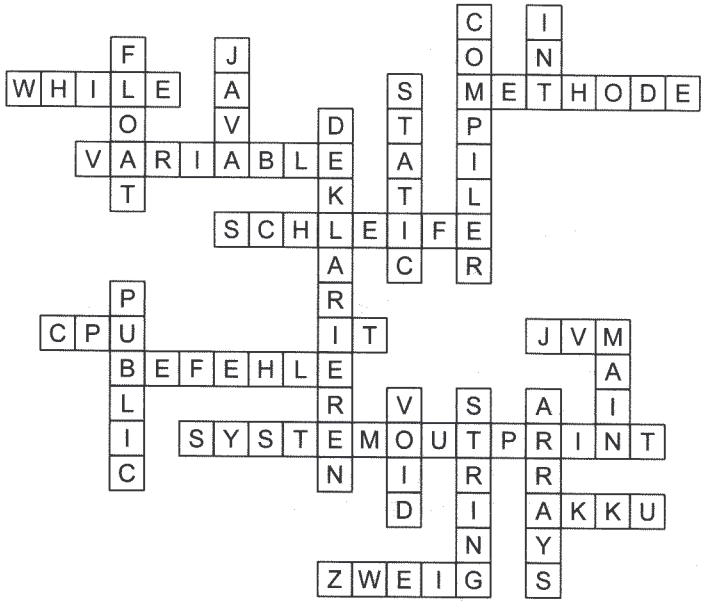
Diese Datei kann ohne eine **Klassendeklaration** nicht kompiliert werden. **Vergessen Sie auch nicht** die schließende geschweifte Klammer!

---

**C**

```
class Übung1b {
  public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("kleines x");
      }
    }
  }
}
```

Der »while«-Schleifencode muss in einer Methode stehen. Er kann nicht einfach so in der Klasse herumhängen.




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

    while ( X < 4 ) {

      System.out.print("da");
      if ( x < 1 ) {
        System.out.print(" ");
      }
      System.out.print("s");

      if ( X > 1 ) {

        System.out.print(" kind");
        x = x + 2;
      }
      if ( x == 1 ) {

        System.out.print("itzend");
      }
      if ( X < 1 ) {

        System.out.print("agt");
      }
      System.out.println("");

      X = X + 1;
    }
  }
}
```

```
Datei Bearbeiten Fenster Hilfe Schummeln
%java PoolPuzzleEins
da sagt
dasitzend
das kind
```

```
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
```

# Ein Ausflug nach Objekthausen

**Man hat mir gesagt, wir hätten es hier mit Objekten zu tun.** Aber in Kapitel 1 haben wir unseren gesamten Code in die main()-Methode gesteckt. Das ist nicht sonderlich objektorientiert. Eigentlich ist das überhaupt nicht objektorientiert! Wir haben zwar ein paar Objekte *gesehen*, wie die String-Arrays für den Phras-O-Mat, aber wir haben keine eigenen Objekt*typen* entwickelt. Jetzt müssen wir diese prozedurale Welt hinter uns lassen, so schnell wie möglich aus main() herauskommen und damit beginnen, eigene Objekte zu machen. Wir werden uns ansehen, was die objektorientierte Entwicklung in Java zu einem solchen Vergnügen macht. Wir werden uns den Unterschied zwischen einer *Klasse* und einem *Objekt* ansehen. Wir werden uns ansehen, wie Objekte Ihr Leben verbessern (zumindest Ihr Programmiererleben – für Ihren Lifestyle sind Sie selbst verantwortlich). Aber seien Sie gewarnt: Ein Ausflug nach Objekthausen ist meist eine Reise ohne Rückkehr. Schicken Sie uns eine Postkarte.

# Stuhlkriege

## (oder wie Objekte Ihr Leben verändern können)

Es war einmal in einer Softwarefirma: Zwei Programmierern wurde die gleiche Spezifikation in die Hand gedrückt, die sie »realisieren« sollten. Der echt nervige Projektmanager zwang die beiden Programmierer zu einem Wettstreit und versprach dem, der die Arbeit zuerst abliefern würde, einen dieser coolen Bürostühle, die all die Typen in den Vorstandsetagen haben. Harry, der prozedurale Programmierer, und Stefan, der OO-Typ, wussten beide, dass das ein Kinderspiel werden würde.

Harry saß in seinem Büro und fragte sich: »Welche Sachen muss dieses Programm *tun*? Welche ***Prozeduren*** brauchen wir?« Und er gab selbst die Antwort: »**rotieren** und **soundAbspielen**.« Was ist denn schon ein Programm, wenn nicht ein Haufen von Prozeduren?

Währenddessen hing Stefan im Café rum und dachte: »Was sind die ***Dinge*** in diesem Programm ... was sind die *Hauptakteure*?« Erst dachte er: **Die Figuren**. Natürlich dachte er auch noch an andere Objekte wie den Benutzer, das Geräusch und das Klick-Ereignis. Aber für diese Teile hatte er bereits eine Codebibliothek. Deswegen konzentrierte er sich darauf, Figuren zu machen. Lesen Sie weiter, damit Sie erfahren, wie Harry und Stefan ihre Programme erstellen, und die Antwort auf die brennende Frage erhalten: ***»Wer bekommt den Stuhl?«***

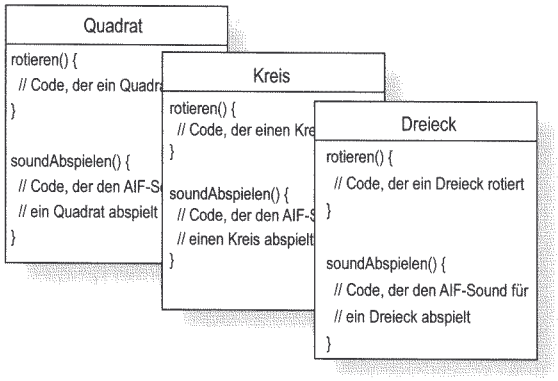
## In Harrys Büro

Wie schon unzählige Male zuvor setzte sich Harry hin, um seine **wichtigen Prozeduren** zu schreiben. Er schrieb **rotieren** und **soundAbspielen** im Handumdrehen.

```
rotieren(figurNr) {
  // die Figur um 360° drehen
}
soundAbspielen(figurNr) {
  // figurNr verwenden, um die
  // AIF-Sounddatei zu suchen,
  // und diese abspielen
}
```

## Auf Stefans Laptop im Café

Stefan schrieb eine ***Klasse*** für jede der drei Figuren.



## Harry dachte, er hätte alles im Griff. Schon fühlte er den glänzenden Stahl des Designerstuhls unter seinem ...

## Einen Augenblick! Es gab eine Änderung in der Spezifikation.

»Gut, *eigentlich* waren Sie der Erste, Harry,« sagte der Manager, »aber wir müssen dem Programm noch eine winzige Kleinigkeit hinzufügen. Für zwei Programmierer-Asse wie Sie ist das bestimmt kein Problem.«

*»Würde ich bloß jedes Mal, wenn ich das höre, nur einen Cent bekommen ...«*, dachte Harry, der wusste, dass die problemlose Spezifikationsänderung ein Hirngespinst war. *»Aber Stefan sieht seltsam locker aus. Was ist da los?«* Harry stand immer noch felsenfest zu seiner Grundüberzeugung, dass der OO-Weg zwar putzig, aber einfach zu langsam sei. Und wenn man diese Überzeugung ändern wollte, dann müsste man sie schon aus seinen kalten, toten, eingeschlafenen Klauen reißen.

← was der Spezifikation hinzugefügt wurde

## Zurück in Harrys Büro

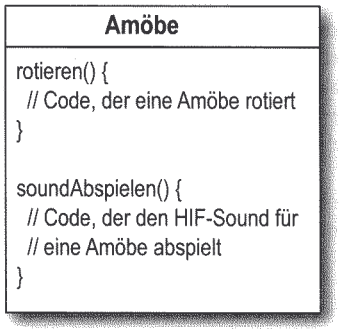
Die rotieren-Prozedur würde immer noch funktionieren. Der Code verwendet eine Lookup-Tabelle, um einer figurNr die richtige grafische Darstellung zuzuordnen. Aber ***soundAbspielen würde sich ändern müssen.*** Und was zum Teufel ist eine .hif-Datei?

```
soundAbspielen(figurNr) {
  // wenn die Figur keine Amöbe ist,
    // figurNr verwenden, um den AIF-Sound
    // zu suchen und abzuspielen,
  // sonst
    // Amöben-.hif-Sound abspielen
}
```

Scheint also doch keine so komplizierte Angelegenheit gewesen zu sein. Aber ***ein mulmiges Gefühl kam doch in ihm hoch, als er bereits getesteten Code anpacken musste.*** *Wenn es irgendwer* wissen musste, dann war *er* es doch – egal was der Projektmanager erzählte, ***Spezifikationen ändern sich immer.***

## Auf Stefans Laptop am Strand

Stefan grinste, nippte an seiner Margarita und schrieb *eine neue Klasse.* Es gab Zeiten, da liebte er an OO am meisten, dass er Code, den er schon getestet und ausgeliefert hatte, nicht mehr anrühren musste. »Flexibilität, Erweiterbarkeit, ...«, dachte er, sich die Vorteile von OO ins Gedächtnis rufend.



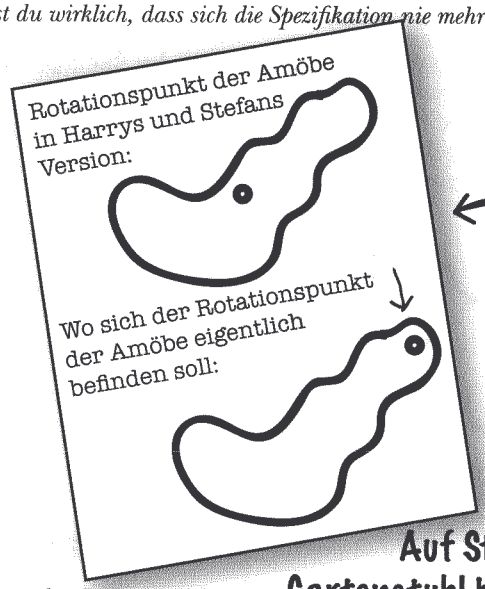
## Harry kam genau einen Augenblick vor Stefan rein

(Haha! Und so viel zu dem wirren OO-Unfug.) Aber das Grinsen auf Harrys Gesicht gefror, als der echt nervige Projektmanager (in diesem Ton aufgesetzter Enttäuschung) sagte: »Ah. Nein. *So* sollte die Amöbe nicht rotieren ...«

Es stellte sich heraus, dass beide Programmierer ihren rotieren-Code so geschrieben hatten:

1. **Ermittle das Rechteck, das die Figur einschließt.**
2. **Berechne den Schwerpunkt dieses Rechtecks und rotiere die Figur um diesen Punkt.**

Aber die Amöbenfigur sollte um einen Punkt an einem *Ende* rotieren wie ein Uhrzeiger. »Ich bin erledigt«, dachte Harry und fühlte sich selbst wie eine niedere Lebensform. »Obwohl, hmmmm. Ich könnte einfach noch ein if/else einbauen und den Rotationspunkt für die Amöbe dann hartcodieren. Das würde wahrscheinlich nichts kaputtmachen.« Aber die leise Stimme in seinem Kopf flüsterte eindringlich: »*Großer Fehler. Glaubst du wirklich, dass sich die Spezifikation nie mehr ändern wird?*«




**was die Spezifikation freundlicherweise nicht gesagt hat**

## Zurück in Harrys Büro

Er hat sich inzwischen überlegt, dass er der Rotationsprozedur besser Rotationspunkt-Argumente hinzufügt. ***Davon war eine Menge Code betroffen.*** Es musste alles komplett wieder getestet und neu kompiliert werden. Dinge, die vorher funktioniert hatten, funktionierten nicht mehr.

```
rotieren(figurNr, xPt, yPt) {
  // Wenn die Figur keine Amöbe ist,
    // auf Basis eines Rechtecks den
    // Schwerpunkt berechnen und dann
    // rotieren
  // sonst
    // xPt und yPt als Rotationspunkt
    // verwenden und dann rotieren
}
```

## Auf Stefans Laptop in seinem Gartenstuhl beim Bluegrass-Festival

Ohne einen Beat zu verpassen, hat Stefan die rotieren-**Methode** verändert – allerdings nur in der Klasse Amöbe. ***Er musste den getesteten, funktionierenden, kompilierten Code*** für die anderen Teile des Programms ***dabei* nicht anpacken.** Um der Amöbe einen Rotationspunkt zu geben, hat er ein **Attribut** hinzugefügt, das alle Amöben haben werden. Während eines einzigen Bela Fleck-Auftritts hat er das Programm verändert, getestet und das überarbeitete Programm (drahtlos) abgeliefert.

**Amöbe**

```
int xPunkt;
int yPunkt;
rotieren() {
  // Code, der eine Amöbe auf
  // Basis von x und y für die
  // Amöbe rotiert
}
soundAbspielen() {
  // Code, der den HIF-Sound
  // für eine Amöbe abspielt
}
```

## Also hat der OO-Typ Stefan den Stuhl bekommen, oder?

*Nicht so schnell.* Harry hat einen Makel an Stefans Ansatz entdeckt. Und da er sicher war, dass er, wenn er den Stuhl bekäme, auch Claudia aus der Buchhaltung bekommen würde, musste er der Angelegenheit eine Wendung geben.

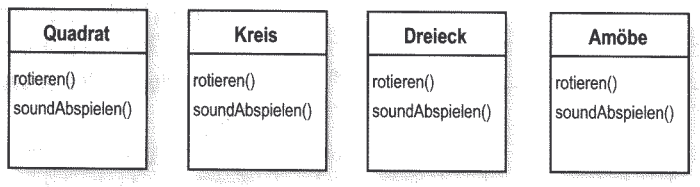
**HARRY:** Du hast da doppelten Code! Die rotieren-Prozedur kommt in allen vier Figur-Dingern vor.

**STEFAN:** Das ist eine ***Methode***, keine *Prozedur*. Und das sind ***Klassen***, keine *Dinger*.

**HARRY:** Ist doch egal. Der Entwurf ist einfach dumm. Du musst *vier* unterschiedliche rotieren-»Methoden« warten. Wie kann das jemals gut sein?

**STEFAN:** Hm. Ich nehme an, du hast den endgültigen Entwurf nicht gesehen. Komm, ich zeig dir, wie OO-**Vererbung** funktioniert, Harry.

Was Harry wollte (in der Annahme, der Stuhl würde sie beeindrucken)

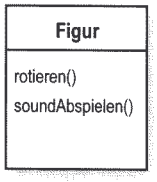



1

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

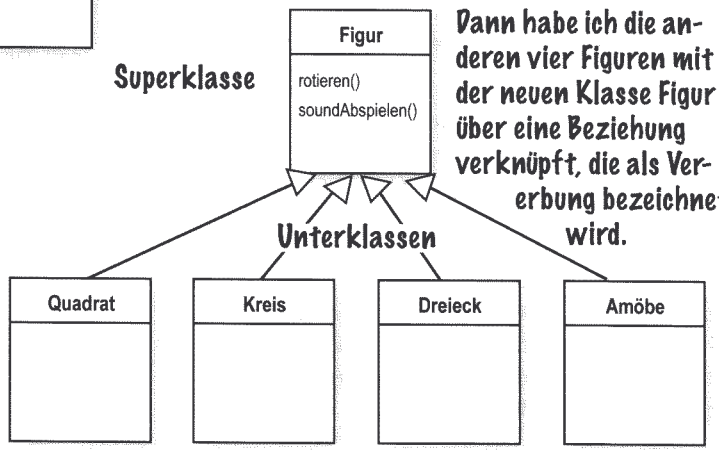
2

Es sind Figuren, und alle rotieren und spielen einen Sound ab. Also habe ich die gemeinsamen Features abstrahiert und in eine neue Klasse gepackt, die ich Figur nenne.




3

Dann habe ich die anderen vier Figuren mit der neuen Klasse Figur über eine Beziehung verknüpft, die als Vererbung bezeichnet wird.




Das kannst du folgendermaßen lesen: »**Rechteck erbt von Figur**«, »**Kreis erbt von Figur**« und so weiter. Ich habe rotieren() und soundAbspielen() aus den anderen Figuren entfernt, und jetzt muss nur noch eine Kopie gewartet werden.
Die Klasse Figur wird als **Superklasse** der anderen vier Klassen bezeichnet. Die anderen vier sind die **Unterklassen** von Figur. Die Unterklassen erben alle Methoden ihrer Superklasse. Anders gesagt: *Wenn die Klasse Figur eine Funktionalität besitzt, erben die Unterklassen automatisch diese Funktionalität.*

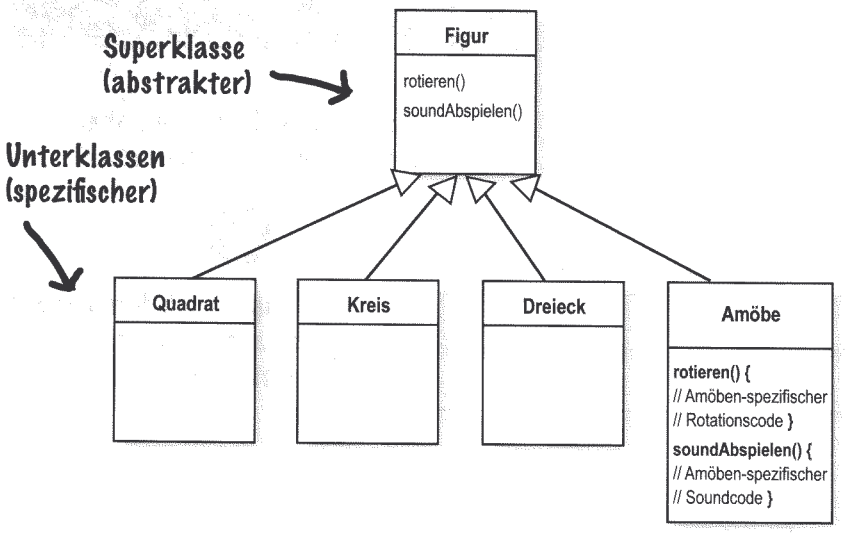
## Und was ist mit dem rotieren() von Amöbe?

**HARRY**: War das nicht gerade das Problem, das wir hatten – dass die Figur Amöbe ganz andere rotieren- und soundAbspielen-Prozeduren hatte?

**STEFAN**: **Methoden.**

**HARRY**: Jaja. Wie kann Amöbe etwas anderes machen, wenn sie ihre Funktionalität von der Klasse Figur »erbt«?

**STEFAN**: Das ist der letzte Schritt. Die Klasse Amöbe **überschreibt** die Methoden der Klasse Figur. Zur Laufzeit weiß die JVM dann genau, welche rotieren()-Methode ausgeführt werden soll, wenn irgendwer der Amöbe sagt, dass sie rotieren soll.




**4**

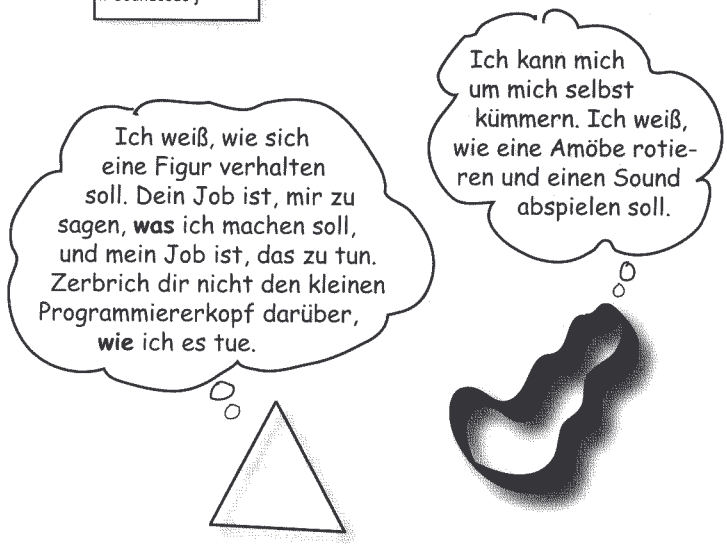
**Ich habe die Klasse Amöbe die rotieren()-Methode der Superklasse Figur überschreiben lassen.**

**Überschreiben heißt, dass eine Unterklasse eine der geerbten Methoden neu definiert, wenn sie das Verhalten dieser Methode ändern oder erweitern muss.**

**Überschreibende Methoden**

**HARRY:** Wie »sagst« du einer Amöbe, dass sie etwas machen soll? Musst du nicht die Prozedur, Entschuldigung – *Methode*, aufrufen und der dann sagen, *was* rotiert werden soll?

**STEFAN:** Das ist ja gerade das, was an OO so cool ist. Wenn zum Beispiel das Dreieck rotiert werden soll, wird durch den Programmcode die rotieren()-Methode *auf dem Dreieck-Objekt* ausgeführt (aufgerufen). Der Rest des Programms hat keine Ahnung davon und kein Interesse daran, *wie* das Dreieck das macht. Und wenn du dem Programm etwas Neues hinzufügen musst, schreibst du einfach eine neue Klasse für den neuen Objekttyp, damit **die neuen Objekte ihr eigenes Verhalten haben**.



## Die Spannung bringt mich um. Wer hat den Stuhl bekommen?

**Hermine aus dem zweiten Stock.**

(Insgeheim hatte der Projektmanager die Spezifikation *drei* Programmierern gegeben.)

## Was magst du an OO?

»OO hilft mir, meine Entwürfe auf natürlichere Weise zu gestalten. Solche Sachen 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.«
- Stefan, 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.«
- Christoph, 39, Projektmanager

»Unglaublich, dass ausgerechnet Christoph das sagen muss. Er hat seit fünf Jahren keine einzige Zeile Code mehr geschrieben.«
- Hans, 44, arbeitet für Christoph

»Außer dem Stuhl?«
- Hermine, 34, Programmiererin

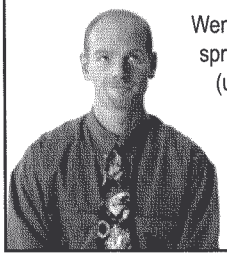
**Zeit, ein paar Neuronen zu massieren.**

Sie haben gerade eine Geschichte über einen prozeduralen Programmierer gelesen, der sich mit einem OO-Programmierer messen musste. Sie haben einen kurzen Überblick über einige OO-Schlüsselkonzepte erhalten, Klassen und Methoden beispielsweise. 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 früherer Arbeit mit anderen OO-Sprachen wissen), und nehmen Sie sich einen Augenblick Zeit, um über die folgenden Fragen nachzudenken:

Auf welche grundlegenden Dinge müssen Sie achten, wenn Sie eine Java-Klasse entwerfen? Welche Fragen müssen Sie sich stellen? Welche Dinge würden Sie in eine Checkliste aufnehmen, die Sie verwenden könnten, wenn Sie eine neue Klasse entwerfen?

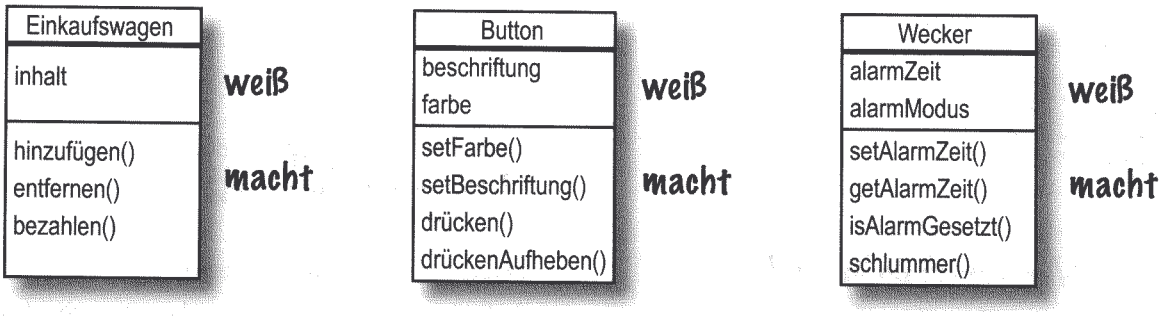
### Lerntipp



Wenn Sie bei einer Übung hängen bleiben, sprechen Sie die Fragen laut aus. Sprechen (und Hören) aktiviert einen anderen Bereich Ihres Gehirns. Das funktioniert allerdings am besten, wenn Sie die Frage mit einer anderen Person diskutieren können. Als Ersatz ist auch ein Haustier geeignet. So hat unser Hund Polymorphie gelernt.

## 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 macht




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

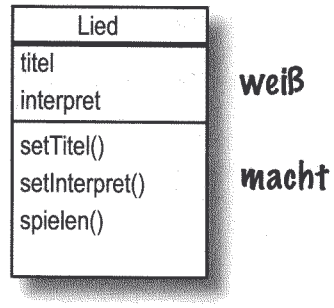
- Instanzvariablen

## Dinge, die ein Objekt *tun* kann, heißen

- Methoden

**Instanzvariablen** (Zustand)

**Methoden** (Verhalten)




Dinge, die ein Objekt über sich selbst *weiß*, werden **Instanzvariablen** genannt. Sie repräsentieren den Zustand eines Objekts (die Daten) und können bei jedem Objekt dieses Typs eindeutige Werte haben.

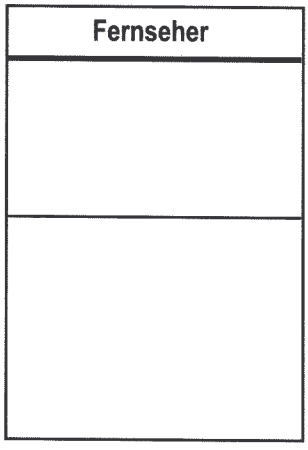
**Betrachten Sie Instanz als eine andere Möglichkeit, Objekt zu sagen.**

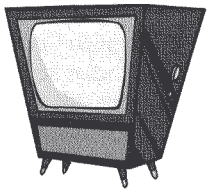
Dinge, die ein Objekt *tun* kann, werden als **Methoden** bezeichnet. Wenn Sie eine Klasse entwerfen, überlegen Sie, welche Daten ein Objekt über sich selbst wissen muss, und entwerfen außerdem die Methoden, die auf diesen Daten operieren. Häufig besitzen Objekte Methoden, die die Werte ihrer Instanzvariablen lesen oder schreiben. Wecker-Objekte haben beispielsweise eine Instanzvariable, die die Weckzeit festhält, und zwei Methoden, um diese Zeit zu setzen und abzufragen.

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

### Spitzen Sie Ihren Bleistift

Fügen Sie ein, was ein Fernseher-Objekt wissen und tun muss.

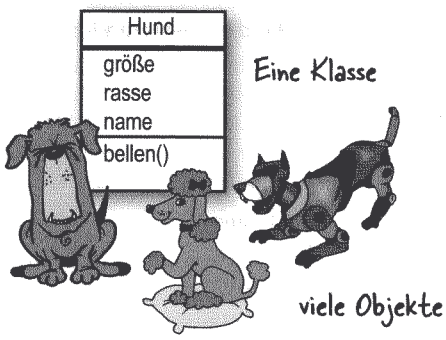


**Instanzvariablen**

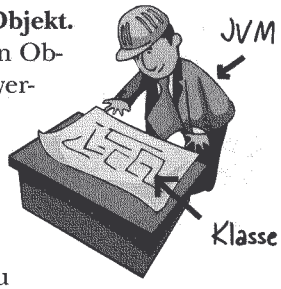
**Methoden**

# Was ist der Unterschied zwischen einer Klasse und einem Objekt?



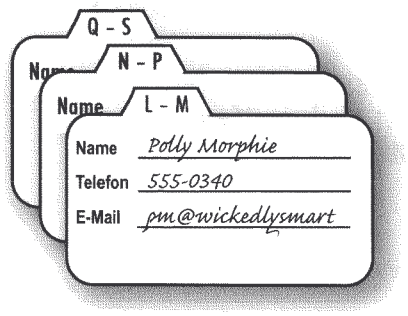

## Eine Klasse ist kein Object.

### (sondern wird verwendet, um es aufzubauen)




**Eine Klasse ist eine *Blaupause* für ein Objekt.** Sie sagt der virtuellen Maschine, *wie* ein Objekt dieses bestimmten Typs gemacht werden soll. Jedes Objekt, das von dieser Klasse gemacht wird, kann eigene Werte für die Instanzvariablen dieser Klasse haben. Beispielsweise könnten Sie die Klasse Button verwenden, um Dutzende unterschiedlicher Buttons zu machen, die alle verschiedene Farben, Größen, Labels und so weiter haben.

## Sehen Sie es mal so ...




**Ein Objekt ist wie ein Eintrag in Ihr Adressbuch.**

Als Analogie für Objekte kann ein Stapel ungebrauchter Karteikarten dienen. Jede Karte hat die gleichen leeren Felder (die Instanzvariablen). Wenn Sie eine Karte ausfüllen, erstellen Sie eine Instanz (ein Objekt), und die Einträge, die Sie auf der Karte machen, repräsentieren den Zustand der Instanz.

Die Methoden der Klasse sind die Dinge, die Sie mit einer bestimmten Karte tun. getName(), setName() und ändereName() könnten alles Methoden für die Klasse Karteikarte sein.

Jede Karte kann also die gleichen Dinge *tun* (getName( ), setName( ) usw.), aber jede Karte *weiß* Dinge, die für diese bestimmte Karte spezifisch sind.

# Ein erstes Objekt machen

Was also braucht man, um ein Objekt zu erzeugen und zu verwenden? Sie brauchen *zwei* Klassen: eine Klasse für den Typ von Objekt, den Sie verwenden wollen (Hund, Wecker, Fernseher usw.), und eine andere Klasse, mit der Sie Ihre neue Klasse *testen* können. In die *Testklasse* packen Sie die main()-Methode, und in dieser main()-Methode erstellen Sie Objekte Ihrer neuen Klassen und verwenden diese. Die Testklasse hat nur einen Zweck: die Methoden und Variablen Ihres neuen Objekttyps auszuprobieren.

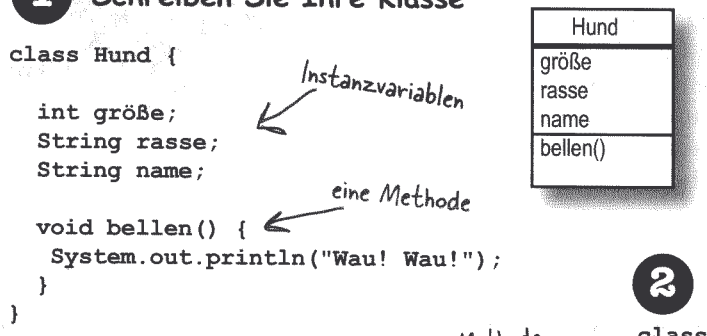
Ab jetzt werden Sie in diesem Buch in vielen Beispielen zwei Klassen sehen: Eine ist die *richtige* Klasse (die Klasse, deren Objekte wir verwenden wollen), und die andere Klasse ist die *Testklasse*, die wir *<IhrXBeliebigerKlassenname>* **`Testlauf`** nennen. Wenn wir eine Klasse **`Bungee`** machen, brauchen wir beispielsweise ebenfalls eine Klasse **`BungeeTestlauf`**. Nur die Klasse *<IrgendeinKlassenname>***`Testlauf`** hat eine main()-Methode. Ihr einziger Zweck ist es, Objekte Ihres neuen Typs (nicht der Testklasse) zu erstellen und dann den Punktoperator zu verwenden, um auf die Methoden und Variablen der neuen Objekte zuzugreifen. Das wird anhand der folgenden Beispiele erstaunlich gut deutlich gemacht.

### Der Punktoperator (.)

**Der Punktoperator (.) verschafft Ihnen Zugriff auf den Zustand und das Verhalten eines Objekts (auf Instanzvariablen und Methoden).**

```
// ein neues Objekt machen

Hund h = new Hund();
// dem Hund sagen, dass er
// bellen soll, indem der
// Punktoperator auf der
// Variablen h verwendet wird,
// um bellen() aufzurufen
h.bellen();

// unter Verwendung des
// Punktoperators größe
// setzen
h.größe = 40;
```

## 1 Schreiben Sie Ihre Klasse




```
class Hund {

  int größe;
  String rasse;
  String name;

  void bellen() {
    System.out.println("Wau! Wau!");
  }
}
```

Instanzvariablen

eine Methode

## 2 Schreiben Sie eine Testklasse (Testlauf)

```
class HundTestlauf {
  public static void main (String[] args) {
    // hier kommt der Hund-Testcode hin
  }
}
```

nur eine main-Methode

(im nächsten Schritt werden wir sie mit Code füllen)

## 3 Erstellen Sie in Ihrer Testklasse ein Objekt und greifen Sie auf die Variablen und Methoden des Objekts zu

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

    Hund h = new Hund();
    h.größe= 40;
    h.bellen();
  }
}
```

Ein Hund-Objekt erstellen

Den Punktoperator (.) verwenden, um die Größe des Hundes zu setzen

und seine bellen()-Methode aufzurufen.

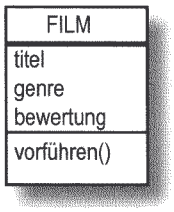
Punkt-operator

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

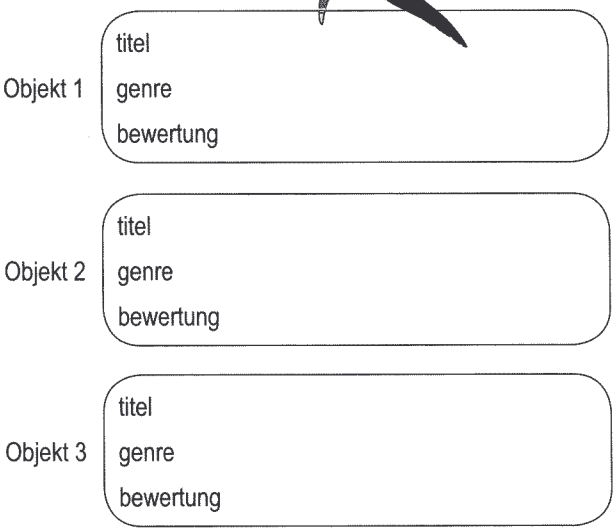
# Film-Objekte machen und testen

```
class Film {
  String titel;
  String genre;
  int bewertung;
  void vorführen() {
    System.out.println("Führe den Film vor");
  }
}

public class FilmTestlauf {
  public static void main(String[] args) {
    Film eins = new Film();
    eins.titel= "Von der Krise verweht";
    eins.genre = "Tragisch";
    eins.bewertung = -2;

    Film zwei = new Film();
    zwei.title = "Vier Deadlines und ein Todesfall";
    zwei.genre = "Komödie";
    zwei.bewertung = 5;
    zwei.vorführen();

    Film drei = new Film();
    drei.titel = "Byte Club";
    drei.genre = "Tragisch, aber letztendlich aufbauend";
    drei.bewertung = 127;
  }
}
```

## Spitzen Sie Ihren Bleistift




Die Klasse FilmTestlauf erzeugt Objekte (Instanzen) der Klasse Film und verwendet den Punktoperator (.), um die Instanzvariablen auf einen bestimmten Wert zu setzen. Die Klasse FilmTestlauf ruft außerdem eine Methode auf einem der Objekte auf. Füllen Sie das Diagramm auf der rechten Seite mit den Werten, die die drei Objekte am Ende von main() haben.



# Schnell! Kehren Sie main den Rücken!

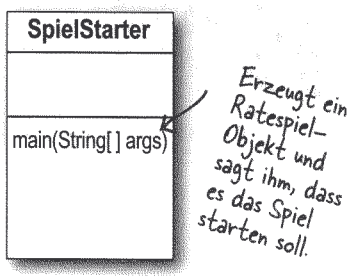
So lange Sie sich in main() befinden, sind Sie noch nicht wirklich in Objekthausen angekommen. Bei einem Testprogramm ist es in Ordnung, wenn es in der main-Methode ausgeführt wird, aber bei einer echten OO-Anwendung brauchen Sie Objekte, die mit anderen Objekten reden, keine statische main()-Methode, die Objekte erstellt und testet.

## Die zwei Verwendungszwecke für main:

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

Eine echte Java-Anwendung besteht aus nichts anderem als aus Objekten, die mit anderen Objekten reden. *Reden* heißt in diesem Fall, dass Objekte aufeinander Methoden aufrufen. Auf der vorangegangenen Seite und in Kapitel 4 können Sie sehen, wie man eine main()-Methode aus einer separaten Testlauf-Klasse verwendet, um die Objekte einer anderen Klasse zu erzeugen und ihre Methoden und Variablen zu testen. In Kapitel 6 sehen wir uns eine Klasse mit einer main()-Methode an, die bei einer *richtigen* Java-Anwendung den Stein ins Rollen bringt (indem sie Objekte macht und diese Objekte dann loslässt, damit sie mit anderen Objekten interagieren usw.).

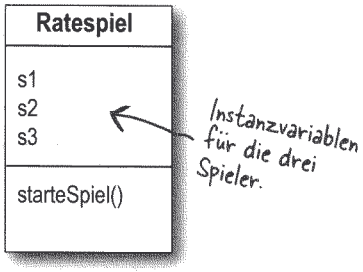
Als Vorschau darauf, wie sich eine echte Java-Anwendung verhalten würde, geben wir Ihnen hier schon einmal ein kleines Beispiel. Weil wir uns immer noch in der ersten Stufe unserer Java-Schule befinden, arbeiten wir bisher mit einem kleinen Werkzeugkasten. Es kann also sein, dass Sie dieses Programm etwas holprig und ineffizient finden. Sie können sich ja überlegen, was Sie tun können, um es zu verbessern. Und das ist dann auch genau das, was wir in späteren Kapiteln tun werden. Machen Sie sich keine Gedanken, wenn Teile des Codes etwas verwirrend sind. Das, worum es in diesem Beispiel geht, ist, dass Objekte mit anderen Objekten reden.




## Das Ratespiel

**Beschreibung:**

Das Spiel umfasst ein Ratespiel-Objekt und drei Spieler-Objekte. Das Spiel generiert eine Zufallszahl zwischen 0 und 9, und die drei Spieler-Objekte versuchen, diese Zahl zu erraten. (Wir haben nie behauptet, dass das ein wirklich *spannendes* Spiel ist.)




**Klassen:**

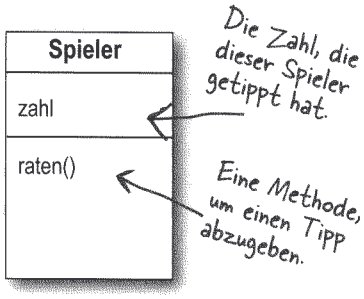
`Ratespiel.class` `Spieler.class` `SpielStarter.class`

**Die Logik:**

1) Die Klasse SpielStarter ist die Klasse, mit der die Anwendung beginnt. Sie hat die main()-Methode.

2) In der main()-Methode wird ein Ratespiel-Objekt erzeugt und seine starteSpiel()-Methode aufgerufen.

3) In der starteSpiel()-Methode des Ratespiel-Objekts wird das vollständige Spiel abgewickelt. Sie erzeugt die drei Spieler und »denkt« sich eine Zufallszahl aus (das von den Spielern zu erratende Ziel). 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 Ratespiel {
  Spieler s1;
  Spieler s2;
  Spieler s3;

  public void starteSpiel() {
    s1 = new Spieler();
    s2 = new Spieler();
    s3 = new Spieler();

    int tipp1 = 0;
    int tipp2 = 0;
    int tipp3 = 0;

    boolean s1hatRecht = false;
    boolean s2hatRecht = false;
    boolean s3hatRecht = false;

    int zielZahl = (int) (Math.random() * 10);
    System.out.println("Ich denke mir eine Zahl zwischen 0 und 9 aus ...");

    while(true) {
      System.out.println("Die zu ratende Zahl ist " + zielZahl);

      s1.raten();
      s2.raten();
      s3.raten();

      tipp1 = s1.zahl;
      System.out.println("Spieler eins tippte " + tipp1);

      tipp2 = s2.zahl;
      System.out.println("Spieler zwei tippte " + tipp2);

      tipp3 = s3.zahl;
      System.out.println("Spieler drei tippte " + tipp3);

      if (tipp1 == zielZahl) {
        s1hatRecht = true;
      }
      if (tipp2 == zielZahl) {
        s2hatRecht = true;
      }
      if (tipp3 == zielZahl) {
        s3hatRecht = true;
      }

      if (s1hatRecht || s2hatRecht || s3hatRecht) {
        System.out.println("Wir haben einen Gewinner!");
        System.out.println("Spieler eins hat richtig geraten? " + s1hatRecht);
        System.out.println("Spieler zwei hat richtig geraten? " + s2hatRecht);
        System.out.println("Spieler drei hat richtig geraten? " + s3hatRecht);
        System.out.println("Das Spiel ist aus.");
        break; // Das Spiel ist zu Ende, also aus der Schleife ausbrechen.

      } else {
        //  Wir müssen weitermachen, weil keiner der Spieler richtig geraten hat!
          System.out.println("Die Spieler müssen es erneut versuchen.");
      } // if/else-Ende
    } // Schleifenende
  } // Methodenende
} // Klassenende
```

Ratespiel hat drei Instanzvariablen für die drei Spieler-Objekte

Erstellt drei Spieler-Objekte und weist sie den Spieler-Instanzvariablen zu

Deklariert drei Variablen, die die Tipps festhalten, die die Spieler abgeben

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

Macht eine »Zielzahl«, die die Spieler erraten müssen

Ruft die raten()-Methode jedes Spielers auf

Ruft den Tipp jedes Spielers ab (das Ergebnis der Ausführung ihrer raten()-Methode), indem auf die zahl-Variable jedes Spielers zugegriffen wird

Prüft die Tipps aller Spieler, um zu sehen, ob einer der Zielzahl entspricht; wenn ein Spieler Recht hat, wird die Variable dieses Spielers auf true gesetzt (denken Sie daran, dass wir sie per Default auf false setzen)

Wenn Spieler eins ODER Spieler zwei ODER Spieler drei Recht hat ... (der ||-Operator bedeutet ODER)

Andernfalls wird die Schleife weiter ausgeführt, und die Spieler werden um einen neuen Tipp gebeten

# Das Ratespiel ausführen

```
public class Spieler {
  int zahl = 0; // hier kommt der Tipp rein

  public void raten() {
    zahl = (int) (Math.random() * 10);
    System.out.println("Ich tippe auf die Zahl: "
                       + zahl);
  }
}

public class SpielStarter {
  public static void main (String[] args) {
    Ratespiel spiel = new Ratespiel();
    spiel.starteSpiel();
  }
}
```

## Java trägt den Müll raus

Jedes Mal, wenn in Java ein Objekt erzeugt wird, kommt es in einen Speicherbereich, der als **Heap** bezeichnet wird. Alle Objekte – ganz gleich wann, wo oder wie sie erzeugt wurden – leben auf dem Heap. Aber das ist nicht einfach irgendein oller Speicher-Heap. Der Java-Heap wird eigentlich als **Garbage Collectible Heap** bezeichnet. Wenn Sie ein Objekt erzeugen, alloziert Java auf dem Heap so viel Speicherplatz, wie das jeweilige Objekt benötigt. Ein Objekt mit 15 Instanzvariablen wird wahrscheinlich mehr Speicherplatz benötigen als ein Objekt mit nur zwei Instanzvariablen. Aber was passiert, wenn Sie den Speicherplatz wieder freigeben möchten? Wie kriegen Sie das Objekt wieder vom Heap, wenn Sie mit ihm fertig sind? Java verwaltet diesen Speicher für Sie! Wenn die JVM »erkennen« kann, dass ein Objekt nie wieder verwendet wird, wird das Objekt zur *Garbage Collection freigegeben*. Und sollte Ihnen der Speicherplatz ausgehen, wird der Garbage Collector ausgeführt. Er schmeißt unerreichbare Objekte raus und macht den Speicherplatz frei, damit er wieder verwendet werden kann. In späteren Kapiteln werden Sie mehr dazu erfahren, wie das funktioniert.

**Ausgabe (ist bei jeder Ausführung anders)**

```
Datei Bearbeiten Fenster Hilfe Explodieren
%java SpielStarter
Ich denke mir eine Zahl zwischen 0 und 9 aus ...
Die zu ratende Zahl ist 6
Ich tippe auf die Zahl: 1
Ich tippe auf die Zahl: 7
Ich tippe auf die Zahl: 3
Spieler eins tippte 1
Spieler zwei tippte 7
Spieler drei tippte 3
Die Spieler müssen es erneut versuchen.
Die zu ratende Zahl ist 6
Ich tippe auf die Zahl: 4
Ich tippe auf die Zahl: 9
Ich tippe auf die Zahl: 3
Spieler eins tippte 4
Spieler zwei tippte 9
Spieler drei tippte 3
Die Spieler müssen es erneut versuchen.
Die zu ratende Zahl ist 6
Ich tippe auf die Zahl: 1
Ich tippe auf die Zahl: 9
Ich tippe auf die Zahl: 6
Spieler eins tippte 1
Spieler zwei tippte 9
Spieler drei tippte 6
Wir haben einen Gewinner!
Spieler eins hat richtig geraten? false
Spieler zwei hat richtig geraten? false
Spieler drei hat richtig geraten? true
Das Spiel ist aus.
```

## Es gibt keine Dummen Fragen

**F: Was ist, wenn ich globale Variablen und Methoden brauche? Wie mache ich das, wenn alles in einer Klasse sein muss?**

A: In einem Java-OO-Programm gibt es das Konzept »globaler« Variablen und Methoden nicht. In der Praxis kann es jedoch Gelegenheiten geben, in denen Sie möchten, dass eine Methode (oder eine Konstante) für Code verfügbar ist, der in einem beliebigen Teil Ihres Programms läuft. Denken Sie zum Beispiel an die `random()`-Methode im Phras-O-Mat. Das ist eine Methode, die von überall aus aufgerufen werden können soll. Oder was ist mit einer Konstanten wie *Pi*? In Kapitel 10 werden Sie lernen, dass eine Methode, die Sie als `public` und `static` deklarieren, sich ziemlich wie eine »globale« Methode verhält. Jeder Code in einer Klasse Ihrer Anwendung kann auf eine Methode zugreifen, die als `public` und `static` deklariert ist. Und wenn Sie eine Variable mit den Modifiern (oder Modifizierern) `public`, `static` und `final` markieren, haben Sie im Grunde eine global verfügbare *`Konstante`* gemacht.

**F: Und das soll immer noch objektorientiert sein, wenn man trotzdem globale Funktionen und globale Daten erzeugen kann?**

A: Zunächst einmal kommt in Java alles in eine Klasse. So werden die Konstante für *Pi* und die Methode für `random()`, auch wenn beide public und static sind, in der Klasse `Math` definiert. 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 Sie nicht mehrere Instanzen/Objekte haben.

**F: Was *ist* ein Java-Programm? Was liefert man denn nun eigentlich?**

A: Ein Java-Programm ist ein Berg von Klassen (oder zumindest *eine* Klasse). In einer Java-Anwendung muss *eine* der Klassen eine main()-Methode haben, die verwendet wird, um das Programm zu starten. Als Programmierer schreiben Sie also eine oder mehrere Klassen. Und diese Klassen sind das, was Sie ausliefern. Wenn der Benutzer keine JVM hat, müssen Sie auch diese mit den Klassen Ihrer Anwendung zusammenpacken, damit er Ihr Programm ausführen kann. Es gibt eine Reihe von Installer-Programmen, die es Ihnen ermöglichen, Ihre Klassen mit einer Vielzahl von JVMs (beispielsweise für unterschiedliche Plattformen) zu bündeln und alles auf eine CD-ROM zu packen. Der Benutzer kann dann die erforderliche Version der JVM installieren (vorausgesetzt, sie ist auf seinem Gerät noch nicht installiert).

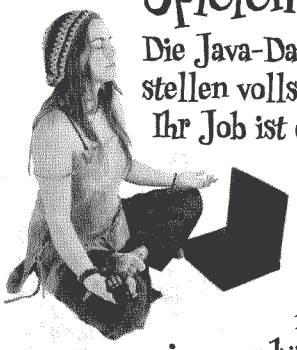
**F: Was ist, wenn ich hundert Klassen habe? Oder tausend? Ist es nicht eine Riesenmühe, all diese Dateien einzeln auszuliefern? Kann ich sie nicht alle zu einem *Anwendungspaket* zusammenschnüren?**

A: Ja, es wäre eine Riesenmühe, Ihren Anwendern einen großen Batzen einzelner Dateien auszuliefern. Aber das müssen Sie nicht tun. Sie können alle Dateien Ihrer Anwendung in ein Java-Archiv stecken – eine *.jar*-Datei –, das auf dem pkzip-Format basiert. In die .jar-Datei können Sie eine einfache Textdatei einschließen, die als etwas formatiert ist, das als *Manifest* bezeichnet wird. Dieses definiert, welche Klasse in der .jar-Datei die main()-Methode enthält, die ausgeführt werden soll.

## Punkt für Punkt

- Bei objektorientierter Programmierung können Sie ein Programm erweitern, ohne bereits getesteten Code anrühren zu müssen.
- Java-Code wird vollständig in **Klassen** definiert.
- Eine Klasse beschreibt, wie ein Objekt dieses Typs gemacht werden soll. **Eine Klasse ist wie eine Blaupause.**
- Ein Objekt kümmert sich um sich selbst – Sie müssen sich keine Gedanken darüber machen und sich auch nicht darum kümmern, *wie* das Objekt es macht.
- Ein Objekt **weiß** Dinge und **tut** Dinge.
- Dinge, die ein Objekt über sich selbst weiß, werden als **Instanzvariablen** bezeichnet. Sie repräsentieren den *Zustand* eines Objekts.
- Dinge, die ein Objekt tut, werden als **Methoden** bezeichnet. Sie repräsentieren das *Verhalten* eines Objekts.
- Wenn Sie eine Klasse erstellen, sollten Sie eventuell auch eine separate Testklasse erstellen, die Sie verwenden, um Objekte Ihres neuen Typs zu erstellen.
- Eine Klasse **erbt** Instanzvariablen und Methoden von einer abstrakteren **Superklasse**.
- Zur Laufzeit ist ein Java-Programm nichts anderes als Objekte, die mit anderen Objekten »reden«.

# Spielen Sie Compiler



Die Java-Dateien auf dieser Seite stellen vollständige Quelldateien dar. Ihr Job ist es, Compiler zu spielen und festzustellen, ob sich diese Dateien kompilieren lassen. Falls nein - wie könnten Sie das in Ordnung bringen? Und falls ja - was käme dabei heraus?

**A**

```
class KassettenRekorder {

  boolean kannAufzeichnen = false;

  void kassetteAbspielen() {
    System.out.println("Spiele Kassette");
  }

  void kassetteAufzeichnen () {
    System.out.println("Zeichne Kassette auf");
  }
}

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

    kr.kannAufzeichnen = true;
    kr.kassetteAbspielen();

    if (kr.kannAufzeichnen == true) {
      kr.kassetteAufzeichnen();
    }
  }
}
```

**B**

```
class DVDPlayer {

  boolean kannAufzeichnen = false;

  void dvdAufzeichnen() {
    System.out.println("DVD aufzeichnen");
  }
}

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

    DVDPlayer d = new DVDPlayer();
    d.kannAufzeichnen = true;
    d.dvdAbspielen();

    if (d.kannAufzeichnen == true) {
      d.dvdAufzeichnen();
    }
  }
}
```

# Code-Magneten

Ein funktionierendes Java-Programm ist vollkommen durcheinander geraten. 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 hinzu, wie Sie benötigen!

```
s.spieleSnare();
```

```
Schlagzeug s = new Schlagzeug();
```

```
boolean tomTom = true;
boolean snare = true;
```

```
void spieleSnare() {
  System.out.println("pamm pamm pa-pamm");
}
```

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

```
if (s.snare == true) {
  s.spieleSnare();
}
```

```
s.snare = false;
```

```
class SchlagzeugTestlauf{
```

```
s.spieleTomTom();
```

```
class Schlagzeug {
```

```
void spieleTomTom () {
  System.out.println("dumm dumm du-dumm");
}
```

```
Datei Bearbeiten Fenster Hilfe Tanzen
% java SchlagzeugTestlauf
pamm pamm pa-pamm
dumm dumm du-dumm
```

# Pool-Puzzle

Ihre ***Aufgabe*** ist es, die Codeschnipsel aus dem Pool zu nehmen und sie in die Lücken im Code einzusetzen. Jedes Schnipsel darf **nur einmal** verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Ihr ***Ziel*** ist es, eine Klasse zu erstellen, die sich kompilieren und ausführen lässt und die die hier gezeigte Ausgabe erzeugt.

**Ausgabe**

```
Datei Bearbeiten Fenster Hilfe Implodieren
%java EchoTestlauf
Halloooo...
Halloooo...
Halloooo...
Halloooo...
10
```

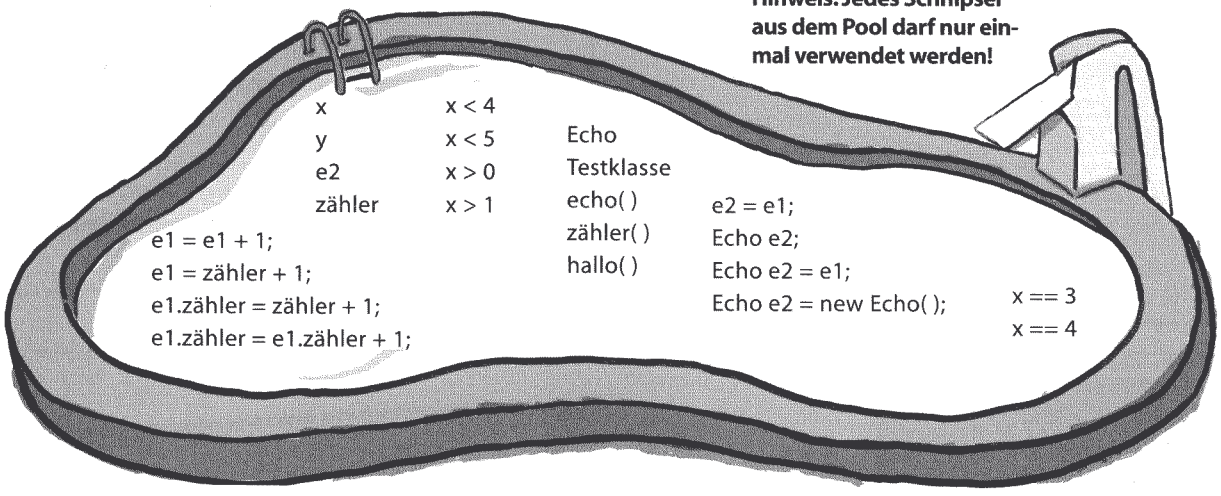
```
public class EchoTestlauf {
  public static void main(String [] args) {
    Echo e1 = new Echo();
    _____________________________
    int x = 0;
    while ( _____________ ) {
      e1.hallo();
      _____________________________
      if ( _____________ ) {
        e2.zähler = e2.zähler + 1;
      }
      if ( _____________ ) {
        e2.zähler = e2.zähler + e1.zähler;
      }
      x = x + 1;
    }
    System.out.println(e2.zähler);
  }
}
```

```
class _____________ {
  int _____________ = 0;
  void _____________ {
    System.out.println("Halloooo... ");
  }
}
```

**Bonusfrage!**

Wie würden Sie das Puzzle vervollständigen, wenn die letzte Zeile der Ausgabe **24** statt **10** wäre?

**Hinweis: Jedes Schnipsel aus dem Pool darf nur einmal verwendet werden!**



x
y
e2
zähler

x < 4
x < 5
x > 0
x > 1

Echo
Testklasse
echo( )
zähler( )
hallo( )

e1 = e1 + 1;
e1 = zähler + 1;
e1.zähler = zähler + 1;
e1.zähler = e1.zähler + 1;

e2 = e1;
Echo e2;
Echo e2 = e1;
Echo e2 = new Echo( );

x == 3
x == 4

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

**Heute Abend anwesend:**

**Klasse Methode Objekt Instanzvariable**

**Ich werde aus einer .java-Datei kompiliert.** Klasse

**Die Werte meiner Instanzvariablen können sich von denen meiner Kumpel unterscheiden.**

**Ich verhalte mich wie eine Schablone.**

**Ich mache gern irgendwelche Sachen.**

**Ich kann viele Methoden haben.**

**Ich repräsentiere einen Zustand.**

**Ich habe Verhalten.**

**Ich befinde mich in Objekten.**

**Ich lebe auf dem Heap.**

**Ich werde verwendet, um Instanzen zu erzeugen.**

**Mein Zustand kann sich ändern.**

**Ich deklariere Methoden.**

**Ich kann mich zur Laufzeit ändern.**

# Lösungen zu den Übungen

## Code-Magneten:

```
class Schlagzeug {

  boolean tomTom = true;
  boolean snare = true;

  void spieleTomTom() {
    System.out.println("dumm dumm du-dumm");
  }

  void spieleSnare() {
    System.out.println("pamm pamm pa-pamm");
  }
}

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

    Schlagzeug s = new Schlagzeug();
    s.spieleSnare();
    s.snare = false;
    s.spieleTomTom();

    if (s.snare == true) {
      s.spieleSnare();
    }
  }
}
```

```
Datei Bearbeiten Fenster Hilfe Tanzen
% java SchlagzeugTestlauf
pamm pamm pa-pamm
dumm dumm du-dumm
```

## Spielen Sie Compiler:

A

```
class KassettenRekorder {
  boolean kannAufzeichnen = false;
  void kassetteAbspielen() {
    System.out.println("Spiele Kassette");
  }
  void kassetteAufzeichnen () {
    System.out.println("Zeichne Kassette auf");
  }
}
```

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

    KassettenRekorder kr = new KassettenRekorder( );
    kr.kannAufzeichnen = true;
    kr.kassetteAbspielen();

    if (kr.kannAufzeichnen == true) {
      kr.kassetteAufzeichnen();
    }
  }
}
```

**Wir haben die Schablone, jetzt müssen wir ein Objekt machen!**

B

```
class DVDPlayer {
  boolean kannAufzeichnen = false;
  void dvdAufzeichnen() {
    System.out.println("DVD aufzeichnen");
  }
  void dvdAbspielen( ) {
    System.out.println("DVD abspielen");
  }
}
```

```
class DVDPlayerTestlauf {
  public static void main(String [] args) {
    DVDPlayer d = new DVDPlayer();
    d.kannAufzeichnen = true;
    d.dvdAbspielen();
    if (d.kannAufzeichnen == true) {
      d.dvdAufzeichnen();
    }
  }
}
```

**Ohne eine Methode kann die Zeile d.dvdAbspielen( ); nicht kompiliert werden!**

# Puzzle-Lösungen

## Pool-Puzzle

```
public class EchoTestlauf {
  public static void main(String [] args) {
    Echo e1 = new Echo();
    Echo e2 = new Echo( ); // die richtige Antwort
        - oder -
    Echo e2 = e1; // die Bonus-Antwort!
    int x = 0;
    while ( x < 4 ) {
      e1.hallo();
      e1.zähler = e1.zähler + 1;
      if ( x == 3 ) {
        e2.zähler = e2.zähler + 1;
      }
      if ( x > 0 ) {
        e2.zähler = e2.zähler + e1.zähler;
      }
      x = x + 1;
    }
    System.out.println(e2.zähler);
  }
}
```

---

```
class Echo {
  int zähler = 0;
  void hallo( ) {
    System.out.println("Halloooo... ");
  }
}
```

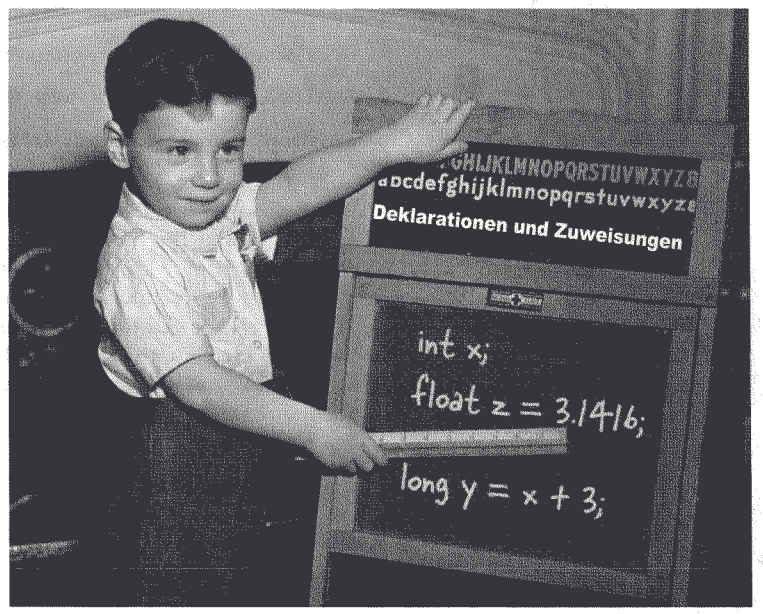
```
Datei Bearbeiten Fenster Hilfe Implodieren
%java EchoTestlauf
Halloooo...
Halloooo...
Halloooo...
Halloooo...
10
```

## Wer bin ich?

| | |
|---|---|
| Ich werde aus einer .java-Datei kompiliert. | Klasse |
| Die Werte meiner Instanzvariablen können sich von denen meiner Kumpel unterscheiden. | Objekt |
| Ich verhalte mich wie eine Schablone. | Klasse |
| Ich mache gern irgendwelche Sachen. | Objekt, Methode |
| Ich kann viele Methoden haben. | Klasse, Objekt |
| Ich repräsentiere einen Zustand. | Instanzvariable |
| Ich habe Verhalten. | Objekt, Klasse |
| Ich befinde mich in Objekten. | Methode, Instanzvariable |
| Ich lebe auf dem Heap. | Objekt |
| Ich werde verwendet, um Instanzen zu erzeugen. | Klasse |
| Mein Zustand kann sich ändern. | Objekt, Instanzvariable |
| Ich deklariere Methoden. | Klasse |
| Ich kann mich zur Laufzeit ändern. | Objekt, Instanzvariable |

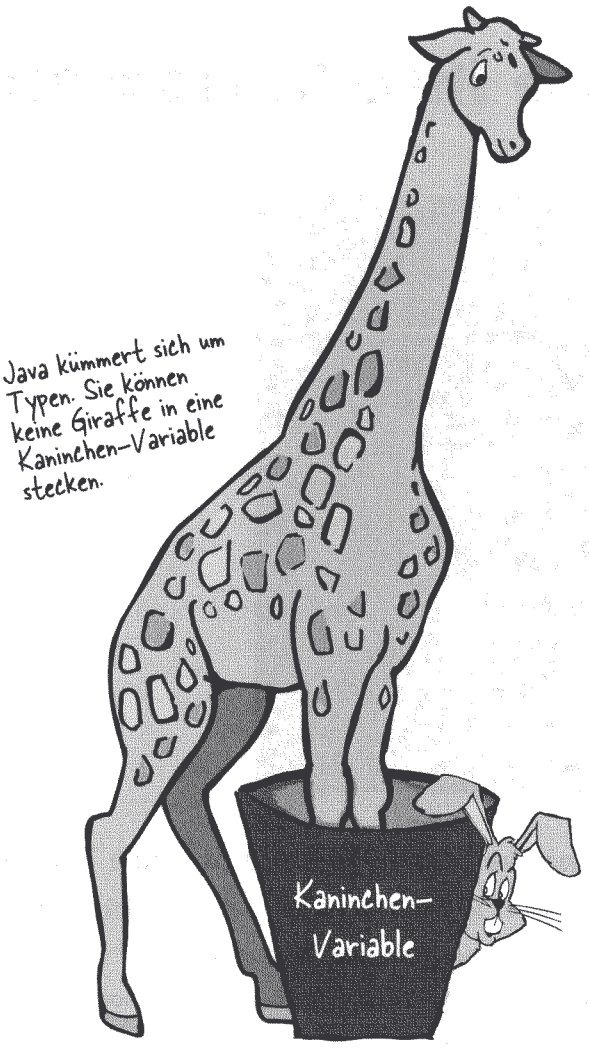
Hinweis: Bei Klassen und Objekten spricht man davon, dass sie Zustand und Verhalten haben. Beides wird in der Klasse definiert, aber man sagt auch, dass das Objekt sie »hat«. Im Augenblick kümmern wir uns nicht darum, wo sie technisch gesehen wirklich leben.

# Verstehen Sie Ihre Variablen




**Variablen gibt es in zwei Geschmacksrichtungen: elementare Typen und Referenztypen.** Bisher haben Sie Variablen an zwei Orten verwendet – als Objekt-**Zustand** (Instanzvariablen) und als **lokale** Variablen (Variablen, die innerhalb einer Methode deklariert werden). Später werden wir Variablen als **Argumente** (Werte, die von aufrufendem Code an eine Methode übergeben werden) und als **Rückgabewerte** (Werte, die aus einer Methode an aufrufenden Code zurückgeliefert werden) verwenden. Sie haben Variablen gesehen, die als einfache, **elementare** Integer-Werte (Typ int) deklariert wurden. Sie haben auch Variablen gesehen, die als etwas **Komplexeres** wie ein String oder ein Array deklariert wurden. Aber **im Leben muss es schließlich mehr geben** als Integer, Strings und Arrays. Was ist, wenn Sie ein HaustierBesitzer-Objekt mit einer Hund-Instanzvariablen haben? Oder ein Auto mit einem Motor? In diesem Kapitel werden wir die Mysterien von Java-Typen offen legen und uns ansehen, was Sie als Variable *deklarieren* können, was Sie in eine Variable *stecken* können und was Sie mit einer Variablen *tun* können. Und schließlich werden wir verstehen, wie das Leben, das sich auf dem Garbage Collectible Heap abspielt, *wirklich* ist.

hier fängt ein neues Kapitel an




# Eine Variable deklarieren

**Java kümmert sich um Typen.** Es erlaubt Ihnen nicht, irgendwelche seltsamen und gefährlichen Dinge zu tun, wie eine Giraffe-Referenz in eine Kaninchen-Variable zu stecken – was passiert, wenn jemand das angebliche Kaninchen auffordert zu `hoppeln()`? Und es erlaubt Ihnen nicht, eine Fließkommazahl in eine Ganzzahlvariable zu packen, es sei denn, Sie *bestätigen dem Compiler*, dass Sie wissen, dass Sie dadurch an Genauigkeit verlieren könnten (alles nach dem Dezimaltrenner beispielsweise).

Der Compiler kann die meisten Probleme ausmachen:

```
Kaninchen klopfer = new Giraffe();
```

Erwarten Sie nicht, dass sich das kompilieren lässt. *Zum Glück.*

Damit dieser ganze Typsicherheitskrempel funktioniert, müssen Sie den Typ Ihrer Variablen deklarieren. Ist es ein int? Ein Hund? Ein einzelnes Zeichen? Variablen gibt es in zwei Geschmacksrichtungen: ***Es gibt elementare (primitive) Typen*** und ***Objektreferenzen.*** Elementare Typen enthalten elementare Werte (denken Sie sich: einfache Bit-Muster) wie Ganzzahlen, Boolesche Werte und Fließkommazahlen. Objektreferenzen halten, genau das, *Referenzen* auf *Objekte* (wer hätte das gedacht!).

Wir werden uns zunächst elementare Typen ansehen und dann dazu kommen, was eine Objektreferenz wirklich bedeutet. Aber unabhängig vom Typ müssen Sie zwei Deklarationsregeln befolgen:

## Variablen müssen einen Typ haben

Außer einem Typ braucht eine Variable noch einen Namen, damit Sie diesen Namen im Code verwenden können.

## Variablen müssen einen Namen haben

Hinweis: Wenn Sie eine Anweisung wie »ein Objekt vom **Typ X**« sehen, denken Sie sich *Typ* und *Klasse* als Synonyme. (In späteren Kapiteln werden wir das noch etwas verfeinern.)

# »Einen doppelten Espresso bitte. Ach nein, mach lieber ein int daraus.«

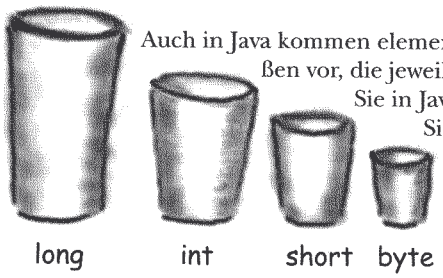
Wenn Sie über Java-Variablen nachdenken, denken Sie an Becher. Kaffee-Becher, Tee-Becher, gigantische Becher mit Massen von Bier, diese großen Becher, in denen man im Kino Popcorn bekommt, Becher mit geschwungenen, sexy Griffen und diese Becher mit Metallverzierungen, die man auf keinen Fall in die Mikrowelle stellen darf.

**Eine Variable ist einfach ein Becher. Ein Behälter. Sie *enthält* etwas.**

Sie hat eine Größe und einen Typ. In diesem Kapitel werden wir uns zunächst die Variablen (Becher) ansehen, die **elementare Typen** enthalten. Etwas später kommen wir zu den Bechern, die *Referenzen auf Objekte* enthalten. Begleiten Sie uns durch die ganze Becher-Analogie. So simpel sie jetzt scheinen mag, sie bietet uns ein vertrautes Mittel, die Dinge zu betrachten, wenn die Diskussion komplexer wird. Und das wird bald passieren.

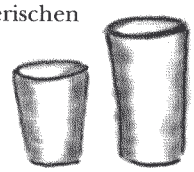
Die Becher für elementare Typen gleichen denen, die Sie im Café bekommen. Wenn Sie schon mal bei Starbucks waren, wissen Sie, was wir meinen. Dort gibt es verschiedene Bechergrößen mit unterschiedlichen Namen wie »Short«, »Tall« und »Ich nehme eine ›Grande‹ Decaf House Blend mit Sahne«.

Vielleicht sind die Becher auf der Theke ausgestellt, damit Sie die richtige Wahl treffen können:




Auch in Java kommen elementare Typen in unterschiedlichen Größen vor, die jeweils unterschiedliche Namen haben. Wenn Sie in Java eine Variable deklarieren, müssen Sie sie mit einem bestimmten Typ deklarieren. Die vier Behälter hier sind für Javas elementare Typen für Ganzzahlen bestimmt.

Jeder Becher enthält einen Wert. Statt »Ich nehme eine Tall Hot Chocolate« sagen Sie dem Compiler bei Javas elementaren Typen deswegen: »Ich nehme eine int-Variable mit dem Wert 90.« Es gibt allerdings einen winzigen Unterschied ... in Java müssen Sie Ihrem Becher auch noch einen *Namen* geben. Eigentlich sagen Sie also: »Ich nehme einen int mit dem Wert 2486. Gib der Variablen bitte den Namen ***höhe***.« Jede Variable eines elementaren Typs besitzt eine festgelegte Anzahl von Bits (die Bechergröße). Die Größen der sechs elementaren numerischen Datentypen von Java sehen Sie hier:




## Elementare Typen

| Typ | Bit-Tiefe | Wertebereich |
|---|---|---|
| **boolean und char** | | |
| boolean | (JVM-spezifisch) | ***true*** oder ***false*** |
| char | 16 Bit | 0 bis 65535 |
| **numerisch (alle vorzeichenbehaftet)** | | |
| ***Ganzzahlen*** | | |
| byte | 8 Bit | -128 bis 127 |
| short | 16 Bit | -32768 bis 32767 |
| int | 32 Bit | -2147483648 bis 2147483647 |
| long | 64 Bit | -gewaltig bis gewaltig |
| ***Fließkommazahlen*** | | |
| float | 32 Bit | variabel |
| double | 64 Bit | variabel |

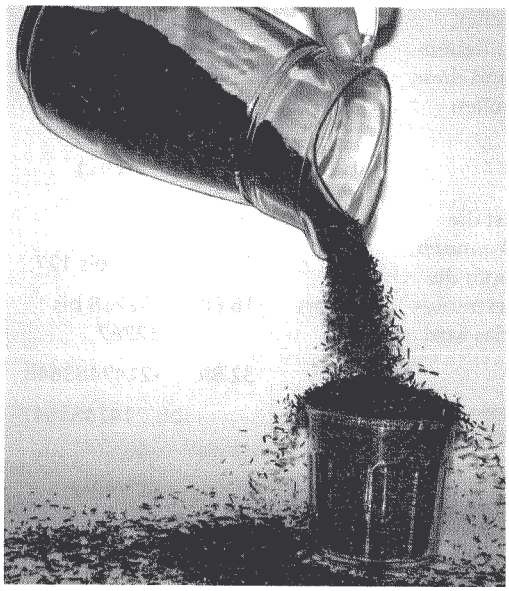
### Deklarationen von elementaren Typen mit Zuweisung:

```
int x;
x = 234;
byte b = 89;
boolean machtSpaß = true;
double d = 3456.98;
char c = 'f';
int z = x;
boolean isPunkRock;
isPunkRock = false;
boolean lichtAn;
lichtAn = machtSpaß;
long big = 3456789;
float f = 32.5f;
```

Achten Sie auf das »f«. Bei einem float-Wert brauchen Sie das, weil Java denkt, dass alles mit einem Dezimaltrenner ein double ist, wenn ihm nicht das 'f' nachgestellt wird.

# Sie wollen doch sicher nichts verschütten ...

**Achten Sie darauf, dass der Wert in die Variable passt.**



Einen großen Wert können Sie nicht in einen kleinen Becher stecken.

Tja, Sie können, aber dann geht etwas verloren. Sie werden dabei etwas *verschütten*. Der Compiler versucht, das zu verhindern, wenn er aus Ihrem Code erkennen kann, dass irgendetwas nicht in den Behälter (Variable/Becher) passt, den Sie verwenden.

Beispielsweise können Sie nicht einfach das Zeug aus einem int in einen Behälter mit byte-Größe stopfen:

```
int x = 24;
byte b = x;
//funktioniert nicht!!
```

Warum das nicht funktioniert, fragen Sie? Schließlich ist der Wert von *x* nur 24, und 24 ist definitiv klein genug, um in ein byte gestopft zu werden. *Sie* wissen das und *wir* wissen das! Der Compiler aber kümmert sich nur darum, dass Sie versuchen, etwas Großes in etwas Kleines zu zwängen, und dabei *eventuell* etwas verschüttet wird. Sie dürfen nicht erwarten, dass der Compiler weiß, welchen Wert *x* hat, selbst wenn Sie ihn in Ihrem Code ausgeschrieben sehen können.

**Einer Variablen können Sie auf mehrere Weisen einen Wert zuweisen, beispielsweise:**

- einen *literalen* Wert nach dem Gleichheitszeichen eingeben (x = ***12***, isGut = ***true*** usw.)
- einer Variablen den Wert einer anderen Variablen zuordnen (x = y)
- in einem Ausdruck Variablen verknüpfen (x = y + ***43***)

In den Beispielen unten werden die literalen Werte fett und kursiv geschrieben:

| | |
|---|---|
| `int größe = 32;` | deklariert einen int namens *größe* und weist ihm den Wert 32 zu |
| `char initial = 'j';` | deklariert einen char namens *initial* und weist ihm den Wert 'j' zu |
| `double d = 456.7;` | deklariert einen double namens *d* und weist ihm den Wert 456,7 zu |
| `boolean isVerrückt;` | deklariert einen boolean namens *isVerrückt* (keine Zuweisung) |
| `isVerrückt = true;` | weist der zuvor deklarierten Variablen *isVerrückt* den Wert *true* zu |
| `int y = x + 456;` | deklariert einen int namens *y* und weist ihm einen Wert zu, der die Summe des aktuellen Werts von *x* plus *456* ist |

## Spitzen Sie Ihren Bleistift

Der Compiler lässt Sie keine Werte aus großen Bechern in kleine Becher stecken. Aber wie sieht das andersherum aus – etwas aus einem kleinen Becher in einen großen schütten? ***Kein Problem.***

Versuchen Sie, auf Grundlage dessen, was Sie über die Größe und die Typen von Variablen mit elementaren Typen wissen, herauszufinden, welche der folgenden Zuweisungen zulässig sind und welche nicht. Wir haben noch nicht alle Regeln behandelt. Bei manchen der Zuweisungen müssen Sie also auf Ihr eigenes Urteilsvermögen setzen. ***Tipp:*** Der Compiler lehnt sich immer in Richtung Sicherheit.

---

***Kreisen*** Sie in der folgenden Liste die Anweisungen ein, die erlaubt wären, wenn diese Zeilen in einer Methode ständen:

```
1. int x = 34.5;
2. boolean boo = x;
3. int g = 17;
4. int y = g;
5. y = y + 10;
6. short s;
7. s = y;
8. byte b = 3;
9. byte v = b;
10. short n = 12;
11. v = n;
12. byte k = 128;
```

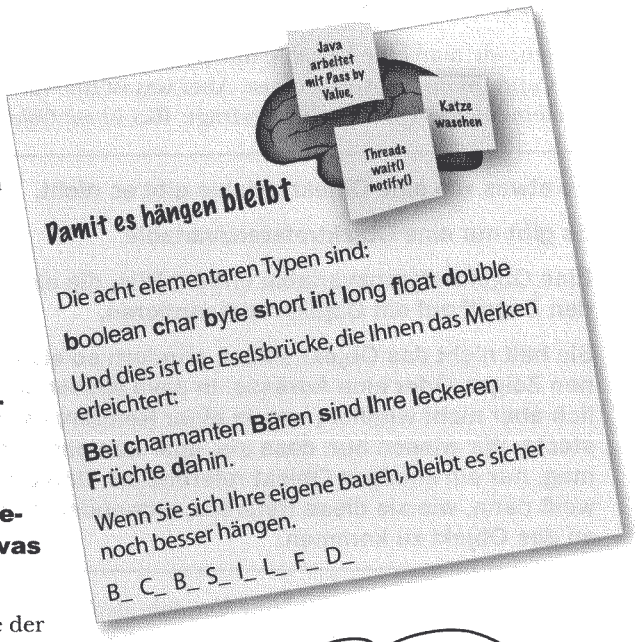
# Finger weg von dem Schlüsselwort!

Sie müssen den Namen und den Typ Ihrer Variablen kennen.

Elementare Typen kennen Sie bereits.

***Aber was können Sie als Name verwenden?*** Die Regeln sind einfach. Sie können Klassen, Methoden und Variablen den folgenden Regeln gemäß benennen (eigentlich sind die Regeln etwas flexibler, aber so bleiben Sie auf der sicheren Seite):

- **Der Name muss mit einem Buchstaben, einem Unterstrich (_) oder einem Dollarzeichen ($) beginnen. Ein Name darf nicht mit einer Zahl beginnen.**
- **Nach dem ersten Zeichen können Sie auch Zahlen verwenden, nur als erstes Zeichen müssen Sie sie vermeiden.**
- **Der Name kann alles sein, was diesen beiden Regeln genügt, darf allerdings nicht einem von Javas reservierten Schlüsselwörtern entsprechen.**




Reservierte Wörter sind Schlüsselwörter (und dergleichen), die der Compiler erkennt. Und wenn Sie Lust haben, Verwirren-wir-den-Compiler zu spielen, dann *versuchen* Sie mal, ein reserviertes Wort als Name zu verwenden.

Einige reservierte Wörter haben Sie bereits gesehen, als wir unsere erste main()-Methode geschrieben haben:

```
public   static   void
```

Keins dieser Wörter dürfen Sie für Ihre eigenen Namen verwenden.

Auch die Namen der elementaren Typen sind reserviert:

```
boolean char byte short int long float double
```

Und es gibt noch jede Menge mehr, die wir bisher nicht besprochen haben. Auch wenn Sie nicht wissen müssen, was sie bedeuten, müssen Sie doch wissen, dass Sie sie selbst nicht verwenden dürfen. ***Versuchen Sie nicht*** *– unter keinen Umständen –,* ***sich die jetzt alle zu merken.*** Um in Ihrem Kopf Platz dafür zu schaffen, müssten Sie wahrscheinlich etwas anderes vergessen. Beispielsweise wo Ihr Auto geparkt ist. Machen Sie sich keine Gedanken. Wenn Sie das Ende des Buchs erreicht haben, sind Ihnen die meisten in Fleisch und Blut übergegangen.

## Hier ist schon reserviert.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| boolean | byte | char | double | float | int | long | short | public | private |
| protected | abstract | final | native | static | strictfp | synchronized | transient | volatile | if |
| else | do | while | switch | case | default | for | break | continue | assert |
| class | extends | implements | import | instanceof | interface | new | package | super | this |
| catch | finally | try | throw | throws | return | void | const | goto | enum |

Javas Schlüsselwörter und andere reservierte Wörter (in keiner irgendwie sinnvollen Anordnung). Wenn Sie eins dieser Wörter als Name verwenden, wirft das den Compiler aus der Bahn.

# Ihr Hund-Objekt steuern

Sie wissen, wie man eine Variable mit einem elementaren Typ deklariert und ihr einen Wert zuweist. Aber was ist mit Variablen für nicht elementare Typen? Anders gefragt: *Was ist mit Objekten?*

- **So etwas wie eine Objektvariable gibt es nicht.**
- **Es gibt nur eine Objektreferenzvariable.**
- **Eine Objektreferenzvariable enthält Bits, die ein Mittel für den Zugriff auf ein Objekt repräsentieren.**
- **Sie hält nicht das Objekt selbst, sondern so etwas wie einen Zeiger. Oder eine Adresse. In Java wissen wir eigentlich aber nicht wirklich, was in einer Referenzvariablen steckt. Wir wissen nur, dass sie, egal was das auch sein mag, nur ein einziges Objekt referenziert. Und die JVM weiß dann, wie sie diese Referenz verwenden muss, um an das Objekt zu kommen.**

Ein Objekt können Sie nicht in eine Variable stecken. Wir stellen uns das oft so vor ... sagen Dinge wie »ich hab den String der Methode System.out.println() übergeben« oder »die Methode liefert einen Hund zurück« oder »ich habe mit new ein Foo-Objekt erzeugt und in die Variable meinFoo gepackt«. Aber das ist nicht, was wirklich passiert. Es gibt keine gigantischen, erweiterbaren Becher, die die Größe jedes beliebigen Objekts annehmen können. Objekte leben immer nur an einem Ort – auf dem Garbage Collectible Heap! (Mehr dazu erfahren Sie weiter unten in diesem Kapitel.)

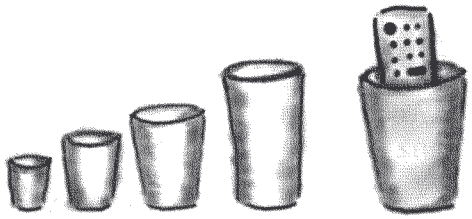
Eine Variable für einen elementaren Typ ist mit Bits gefüllt, die den tatsächlichen ***Wert*** der Variablen repräsentieren. Eine Variable mit einer Objektreferenz ist mit Bits gefüllt, die ***einen Weg repräsentieren, das Objekt zu erhalten.***

Verwenden Sie den Punktoperator (.) auf einer Referenzvariablen, um zu sagen: »Nimm das Ding *vor* dem Punkt, um mir das Ding *nach* dem Punkt zu besorgen.«

```
meinHund.bellen();
```

bedeutet zum Beispiel: »Nimm das Objekt, das die Variable meinHund referenziert, um die Methode bellen() aufzurufen.« Stellen Sie sich die Anwendung des Punktoperators auf eine Objektreferenzvariable so vor, als würden Sie einen Knopf auf einer Fernbedienung für dieses Objekt drücken.

**Hund h = new Hund();**
**h.bellen();**



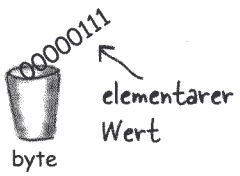

# Eine Objektreferenz ist einfach ein weiterer Variablenwert.

**Etwas, das in einen Becher kommt.**
**Nur ist der Wert diesmal eine Fernsteuerung.**

## Elementare Variable

```
byte x = 7;
```

Die Bits, die 7 repräsentieren, werden in die Variable gesteckt (00000111).

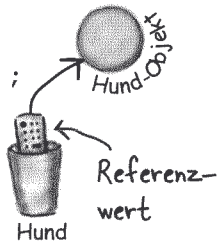



## Referenzvariable

```
Hund meinHund = new Hund();
```

Die Bits, die den Weg zum Zugriff auf das Hund-Objekt repräsentieren, kommen in die Variable.

***Das Hund-Objekt selbst kommt nicht in die Variable!***



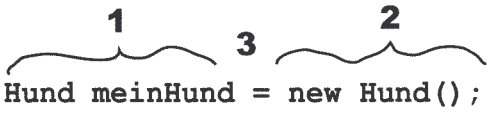

Bei elementaren Variablen ist der Wert der Variablen ... eben der Wert (5, –26.7, 'a').

Bei Referenzvariablen ist der Wert der Variablen ... Bits, die einen Weg repräsentieren, zu einem bestimmten Objekt zu gelangen.

Sie wissen nicht (oder interessieren sich nicht dafür), wie eine bestimmte JVM Objektreferenzen implementiert. Klar. Es könnte da Zeiger auf Zeiger geben. Aber selbst wenn Sie das wissen, können Sie die Bits trotzdem für nichts anderes verwenden als für den Zugriff auf ein Objekt.

Uns ist egal, wie viele 1en und 0en es in einer Referenzvariablen gibt. Das ist den einzelnen JVMs und den Mondphasen überlassen.

## Die 3 Schritte des Deklarierens, Erzeugens und Zuweisens von Objekten




### 1 Eine Referenz deklarieren

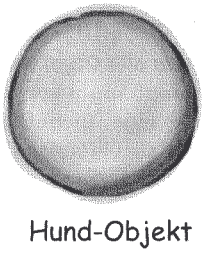
```
Hund meinHund = new Hund();
```

... sagt der JVM, dass sie den Speicherplatz für eine Referenzvariable allozieren soll, und gibt der Variablen den Namen *meinHund*. Die Referenzvariable hat für immer den Typ Hund. Das heißt, sie ist eine Fernsteuerung, die Schalter zur Steuerung eines Hundes hat, aber keine zur Steuerung einer Katze, eines Buttons oder eines Sockets.

### 2 Ein Objekt erzeugen

```
Hund meinHund = new Hund();
```

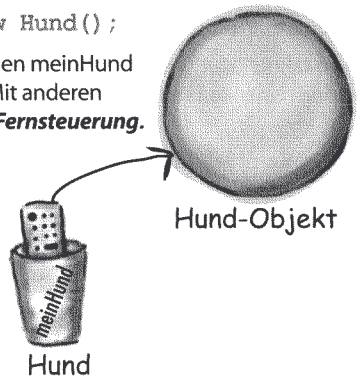
... sagt der JVM, dass sie auf dem Heap Speicherplatz für ein neues Hund-Objekt allozieren soll (über diesen Vorgang werden wir noch eine Menge lernen, insbesondere in Kapitel 9).




### 3 Objekt und Referenz verknüpfen

```
Hund meinHund = new Hund();
```

... weist der Referenzvariablen meinHund das neue Hund-Objekt zu. Mit anderen Worten: ***programmiert die Fernsteuerung.***



## Es gibt keine Dummen Fragen

**F: Wie groß ist eine Referenzvariable?**

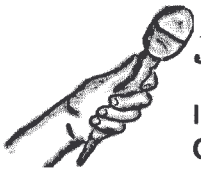
A: Das weiß man nicht. Wenn Sie nicht gerade jemanden aus dem Entwicklungsteam einer JVM zu Ihren Freunden zählen, wissen Sie nicht, wie eine Referenz repräsentiert wird. Irgendwo stecken da Zeiger drin, aber Sie können nicht auf sie zugreifen. Und das brauchen Sie auch nicht. (Na gut! Wenn Sie darauf bestehen. Sie können sie sich als einen 64-Bit-Wert vorstellen.) Aber wenn es um die Allozierung von Speicherplatz geht, sollte die wichtige Frage sein, wie viele *Objekte* (nicht wie viele *Objektreferenzen*) Sie erstellen und wie groß *diese* (die *Objekte*) wirklich sind.

**F: Soll das also heißen, dass alle Objektreferenzen, unabhängig von der Größe des eigentliches Objekts, auf das sie verweisen, die gleiche Größe haben?**

A: Ja. Bei jeder bestimmten JVM haben alle Referenzen die gleiche Größe, und zwar unabhängig davon, welche Größe das Objekt hat, das sie referenzieren. Aber jede JVM kann andere Mittel verwenden, um Referenzen zu repräsentieren. Die Referenzen der einen JVM können also kleiner oder größer als die Referenzen einer anderen JVM sein.

**F: Kann ich auf einer Referenzvariablen arithmetische Operationen vornehmen, sie inkrementieren und so – C-Zeug eben?**

A: Nein. Sprechen Sie mir nach: »Java ist nicht C.«



## Java im Gespräch

### Interview der Woche: Objektreferenz

**Von Kopf bis Fuß:** Möchten Sie uns vielleicht sagen, wie das Leben einer Objektreferenz ist?

**Referenz:** Eigentlich sehr einfach. Ich bin eine Fernsteuerung und kann zur Steuerung unterschiedlicher Objekte programmiert werden.

**Von Kopf bis Fuß:** Heißt das auch unterschiedlicher Objekte, während Sie bereits in Betrieb sind? Ich meine, können Sie auf einen Hund verweisen und dann fünf Minuten später auf ein Auto?

**Referenz:** Natürlich nicht. Wenn ich einmal deklariert bin, ist die Sache gegessen. Wenn ich eine Hund-Fernsteuerung bin, kann ich nie auf etwas anderes als einen Hund zeigen (autsch – wieder dieser Freud'sche Bug, *zeigen* sollen wir eigentlich nicht sagen), etwas anderes als einen Hund *referenzieren*, wollte ich sagen.

**Von Kopf bis Fuß:** Heißt das, dass Sie nur *einen* Hund referenzieren können?

**Referenz:** Nein. Ich kann einen Hund referenzieren und dann fünf Minuten später irgendeinen *anderen* Hund. So lange es ein Hund ist, kann ich auf ihn umgeleitet werden (so wie man eine Fernbedienung für einen anderen Fernseher umprogrammiert). Außer ... nein.

**Von Kopf bis Fuß:** Doch, doch. Sie können es mir ruhig sagen. Was wollten Sie sagen?

**Referenz:** Eigentlich möchte ich mich darauf jetzt lieber nicht einlassen. Aber vielleicht gebe ich Ihnen doch die kurze Version – wenn ich als final markiert bin, kann ich, nachdem ich einem Hund zugewiesen wurde, nie mehr auf etwas anderes umprogrammiert werden als auf *diesen* einen Hund. Mit anderen Worten: Mir kann kein anderes Objekt zugewiesen werden.

**Von Kopf bis Fuß:** Sie haben Recht. Darüber wollen wir jetzt noch nicht sprechen. Aber wenn Sie nicht als final markiert sind, können Sie erst einen Hund und später dann einen anderen Hund referenzieren. Ist es auch möglich, dass Sie irgendwann einmal *überhaupt gar nichts* referenzieren? Ist es möglich, dass Sie auf nichts programmiert sind?

**Referenz:** Ja. Aber darüber rede ich nicht gern.

**Von Kopf bis Fuß:** Warum das?

**Referenz:** Weil das heißt, dass ich `null` bin, und das regt mich auf.

**Von Kopf bis Fuß:** Sie meinen, weil Sie dann keinen Wert haben?

**Referenz:** Ach nein, `null` *ist* ein Wert. Ich bin immer noch eine Fernsteuerung. Aber es ist, als hätten Sie sich eine neue universelle Fernsteuerung gekauft, obwohl Sie keinen Fernseher haben. Ich bin nicht dafür programmiert, irgendetwas zu steuern. Sie können den ganzen Tag meine Knöpfe drücken, und es passiert nichts. Ich fühl mich dann einfach so ... so nutzlos. Eine Verschwendung von Bits. Nicht vieler Bits natürlich. Aber immerhin. Und das ist noch nicht einmal das Schlimmste. Wenn ich die einzige Referenz auf ein bestimmtes Objekt bin und dann auf null gesetzt werde (meine Programmierung gelöscht wird), heißt das, dass *niemand* mehr an das Objekt kommt, das ich referenziert hatte.

**Von Kopf bis Fuß:** Und warum ist das schlecht?

**Referenz:** Muss man das *fragen?* Da habe ich ein Verhältnis zu dem Objekt aufgebaut, eine enge Beziehung, und dann wird dieses Band plötzlich grausam durchtrennt. Und ich werde dieses Objekt nie wiedersehen, weil es jetzt für die [Produzent, unterlegen mit tragischer Musik] *Garbage Collection* freigegeben wird. Schluchz. Aber denkt ihr Programmierer jemals *daran?* Schluchz. Warum, *warum* kann ich kein primitiver Typ sein? *Ich hasse es, eine Referenz zu sein.* Diese Verantwortung, all die zerbrochenen Bindungen ...

## Leben auf dem Garbage Collectible Heap

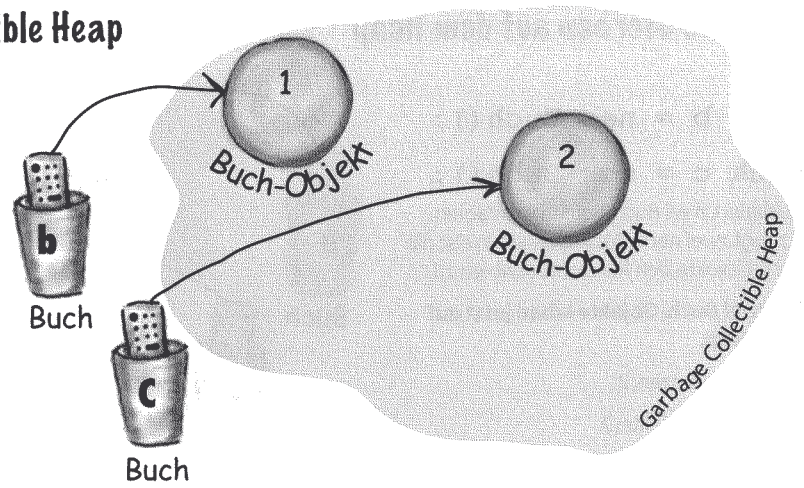
```
Buch b = new Buch();
Buch c = new Buch();
```

Deklariert zwei Buch-Referenzvariablen. Erzeugt zwei neue Buch-Objekte. Weist die Buch-Objekte den Referenzvariablen zu.

**Die beiden Buch-Objekte leben jetzt auf dem Heap.**

Referenzen: 2

Objekte: 2




---

```
Buch d = c;
```

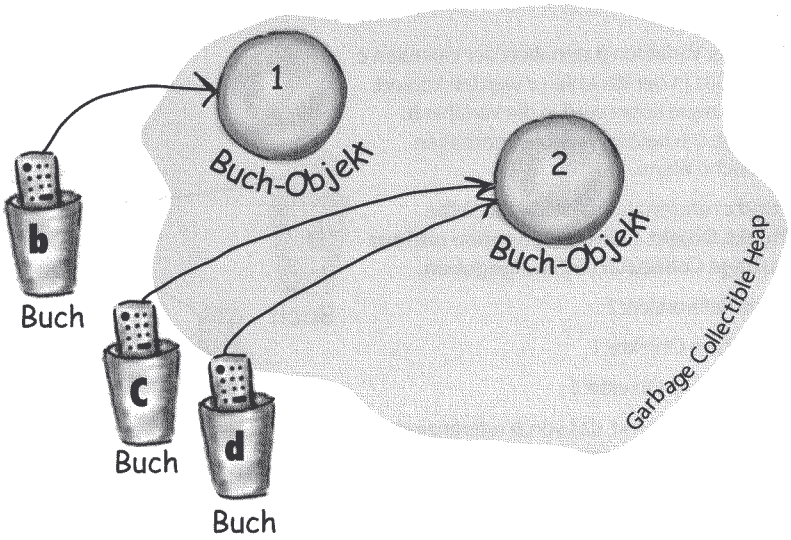
Deklariert eine neue Buch-Referenzvariable. Anstatt ein neues, drittes Buch-Objekt zu erzeugen, wird der Variablen ***d*** der Wert der Variablen ***c*** zugewiesen. Was aber heißt das? Das ist, als sagte man: »Nimm die Bits in ***c***, mach eine Kopie davon und steck diese Kopie in ***d***.«

***c* und *d* verweisen auf dasselbe Objekt.**

**Die Variablen *c* und *d* halten zwei separate Kopien des gleichen Werts. Zwei Fernsteuerungen, die auf einen Fernseher programmiert sind.**

Referenzen: 3

Objekte: 2




---

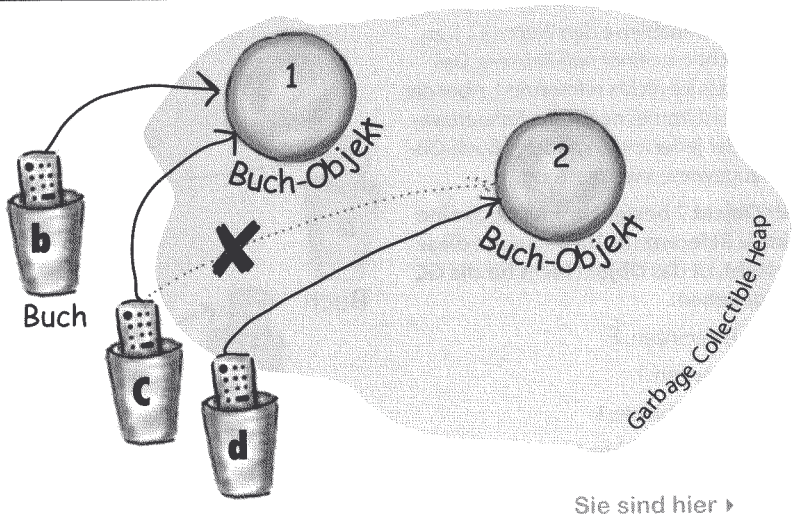
```
c = b;
```

Weist der Variablen ***c*** den Wert der Variablen ***b*** zu. Jetzt wissen Sie bereits, was das heißt. Die Bits in der Variablen ***b*** werden kopiert, und diese neue Kopie wird in die Variable ***c*** gesteckt.

***b* und *c* referenzieren beide dasselbe Objekt.**

Referenzen: 3

Objekte: 2



## Leben und Sterben auf dem Heap

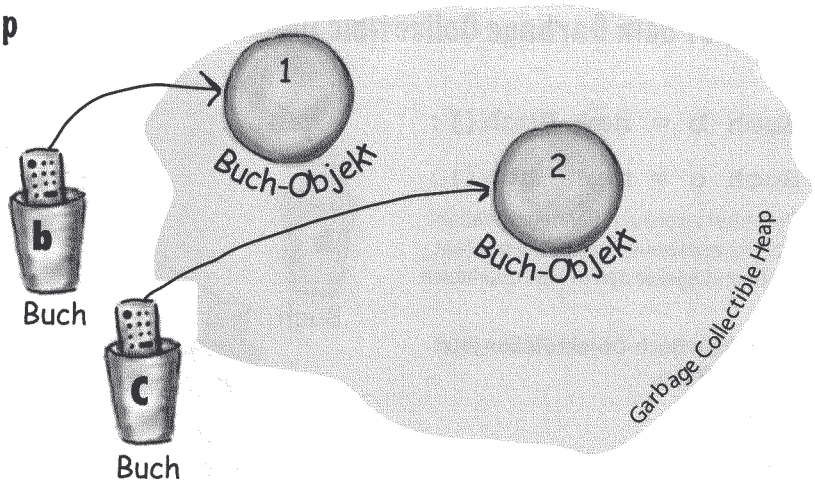
```
Buch b = new Buch();
Buch c = new Buch();
```

Deklariert zwei Buch-Referenzvariablen. Erzeugt zwei neue Buch-Objekte. Weist die Buch-Objekte den Referenzvariablen zu.

**Die zwei Buch-Objekte leben jetzt auf dem Heap.**

Aktive Referenzen: 2

Erreichbare Objekte: 2




---

```
b = c;
```

Weist der Variablen ***b*** den Wert der Variablen ***c*** zu. Die Bits in der Variablen ***c*** werden kopiert, und die neue Kopie wird in die Variable ***b*** gestopft. Die beiden Variablen enthalten identische Werte.

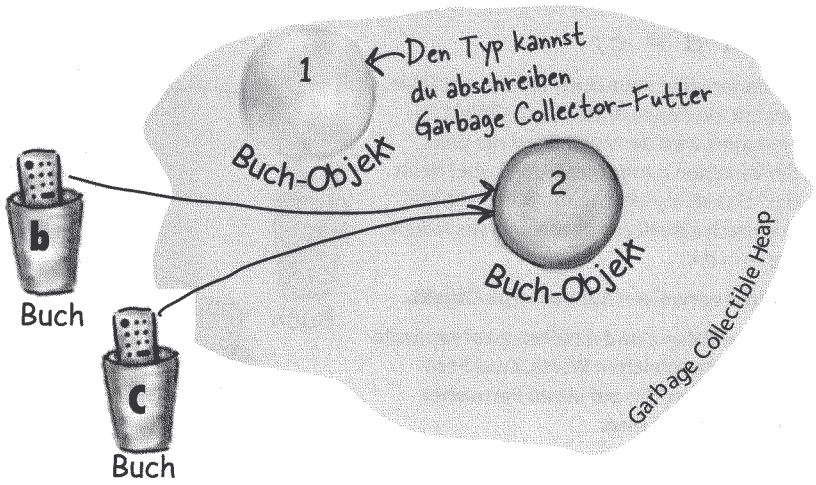
***b* und *c* referenzieren beide dasselbe Objekt. Objekt 1 wird aufgegeben und zur Garbage Collection (GC) freigegeben.**

Aktive Referenzen: 2

Erreichbare Objekte: 1

Aufgegebene Objekte: 1

Für das erste Objekt, das von ***b*** referenziert wurde, Objekt 1, bestehen keine Referenzen mehr. Es ist *unerreichbar*.




---

```
c = null;
```

Weist der Variablen ***c*** den Wert `null` zu. Das macht ***c*** zu einer *null-Referenz*. Das heißt, dass sie nichts referenziert. Aber die Variable ist immer noch eine Referenzvariable, der jederzeit ein anderes Buch-Objekt zugewiesen werden kann.

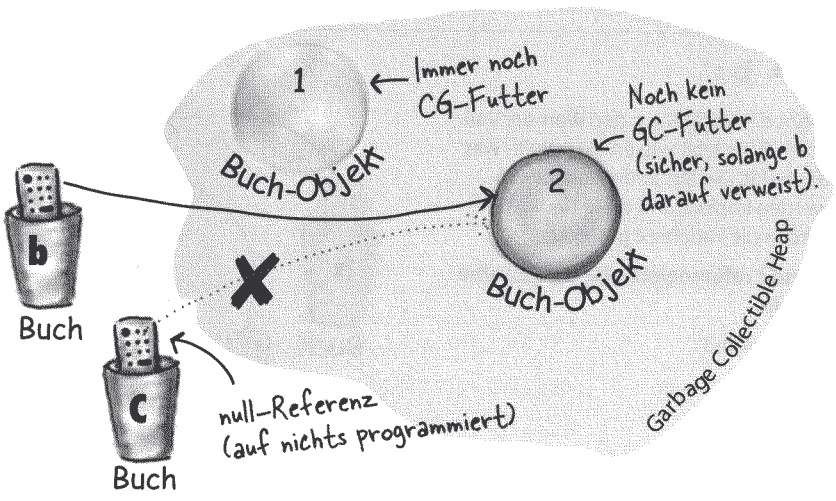
**Für Objekt 2 besteht immer noch eine aktive Referenz (*b*). Und solange diese besteht, ist das Objekt nicht für die GC freigegeben.**

Aktive Referenzen: 1

*null*-Referenzen: 1

Erreichbare Objekte: 1

Aufgegebene Objekte: 1



# Ein Array ist wie ein Tablett mit Bechern

1. Eine int-Array-Variable deklarieren. Eine Array-Variable ist eine Fernsteuerung für ein Array-Objekt.

```
int[] zahlen;
```

2. Ein neues int-Array mit der Länge 7 erzeugen und der zuvor deklarierten `int[]`-Variablen `zahlen` zuweisen.

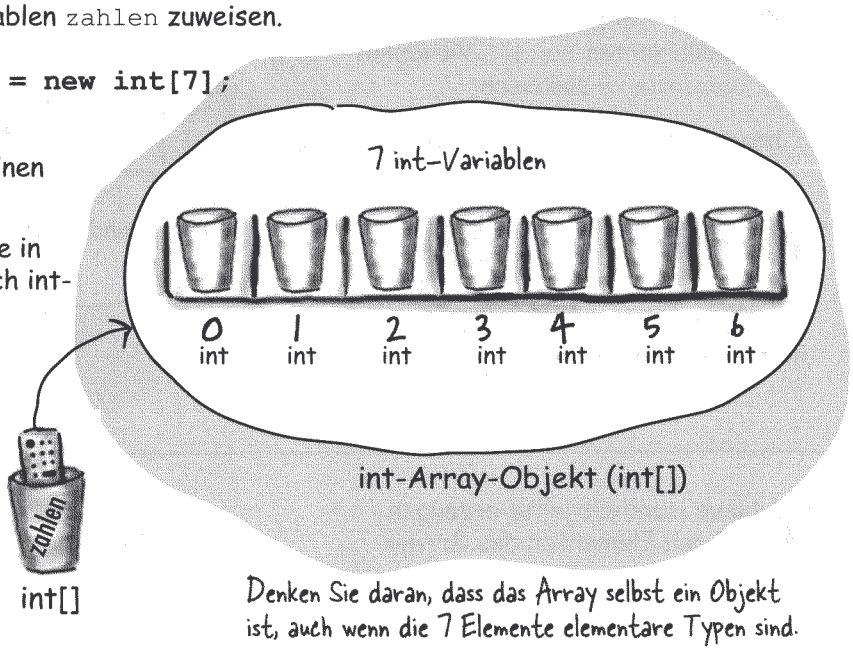
```
zahlen = new int[7];
```

3. Jedem Element im Array einen int-Wert geben.

Denken Sie daran: Elemente in einem int-*Array* sind einfach int-*Variablen*.

```
zahlen[0] = 6;
zahlen[1] = 19;
zahlen[2] = 44;
zahlen[3] = 42;
zahlen[4] = 10;
zahlen[5] = 20;
zahlen[6] = 1;
```

7 int-Variablen




Denken Sie daran, dass das Array selbst ein Objekt ist, auch wenn die 7 Elemente elementare Typen sind.

# Arrays sind auch Objekte

Die Java-Standardbibliothek schließt viele ausgefeilte Datenstrukturen einschließlich Maps, Trees und Sets ein (siehe Kapitel 16). Aber Arrays sind klasse, wenn Sie eine schnelle, geordnete, effiziente Liste für Dinge haben möchten. Arrays bieten Ihnen einen schnellen wahlfreien Zugriff, indem sie Ihnen ermöglichen, über eine Indexposition auf beliebige Elemente im Array zuzugreifen.

Jedes Element in einem Array ist einfach eine Variable – entweder einer der elementaren Typen (denken Sie an Bären und leckere Früchte) oder eine Referenzvariable. Alles, was Sie in eine *Variable* dieses Typs stecken können, kann einem *Array-Element* dieses Typs zugewiesen werden. In einem Array vom Typ int (int[]) kann jedes Element also ein int festhalten. In einem Hund-Array (Hund[]) kann jedes Element ... einen Hund festhalten? Nein. Denken Sie daran, dass eine Referenzvariable einfach nur eine Referenz festhält (eine Fernsteuerung), nicht das Objekt selbst. In einem Hund-Array kann jedes Element also eine *Fernsteuerung* für einen Hund festhalten. Natürlich müssen wir erst noch die Hund-Objekte erzeugen ... und das werden Sie auf der nächsten Seite sehen.

Aber sicher haben Sie das Wichtigste in dem Bild oben erkannt – *das Array ist ein Objekt, auch wenn es ein Array für elementare Typen ist.*

**Arrays sind immer Objekte, egal ob sie laut Deklaration elementare Typen oder Objektreferenzen festhalten sollen.** Aber Sie können ein Array-Objekt haben, dessen Deklaration sagt, dass es elementare Werte *festhält.* Anders gesagt: Das Array-Objekt kann *Elemente* haben, die elementare Typen sind, aber das Array selbst ist *nie* ein elementarer Typ. Egal, was das Array enthält, das Array selbst ist immer ein Objekt!

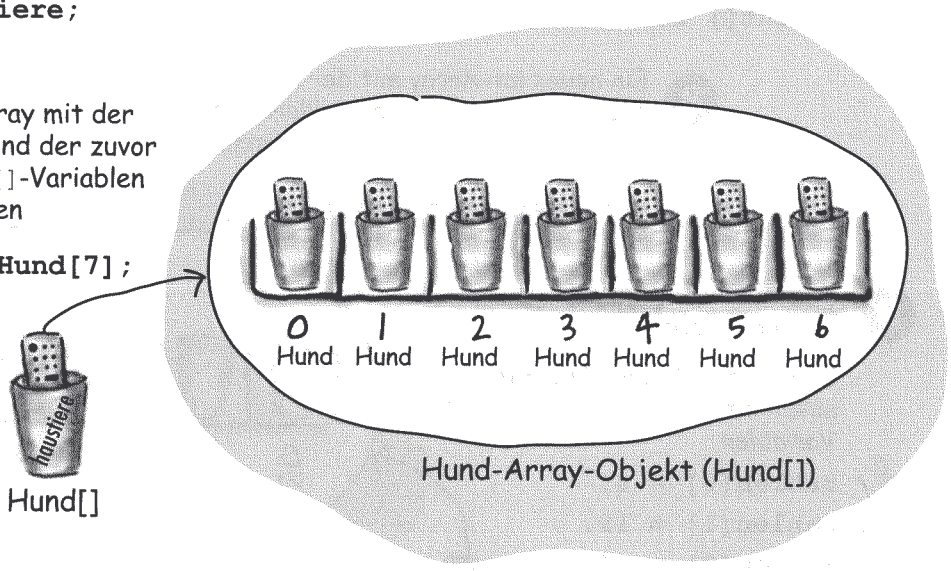
# Ein Hund-Array machen

**1** Eine Hund-Array-Variable deklarieren

```
Hund[] haustiere;
```

**2** Ein neues Hund-Array mit der Länge 7 erzeugen und der zuvor deklarierten `Hund[]`-Variablen `haustiere` zuweisen

```
haustiere = new Hund[7];
```

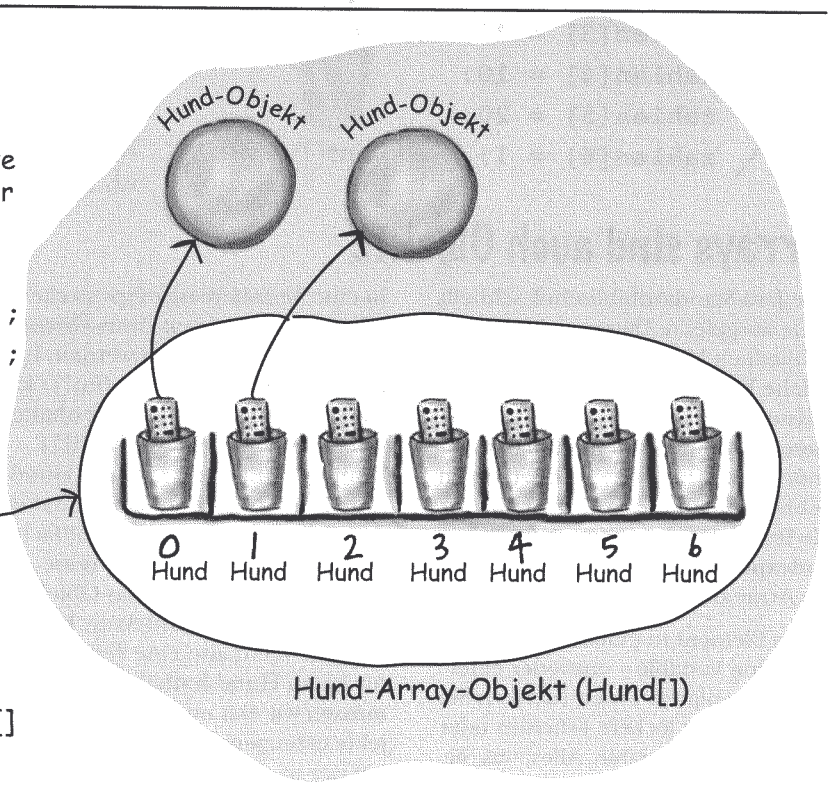



## Was fehlt?

**Hunde! Wir haben ein Array mit Hund-*Referenzen*, aber noch keine Hund-*Objekte*!**

**3** Neue Hund-Objekte erzeugen und den Elementen des Arrays zuweisen
Denken Sie daran, die Elemente im Hund-*Array* sind einfach nur Hund-Referenz*variablen*. Wir brauchen immer noch Hunde!

```
haustiere[0] = new Hund();
haustiere[1] = new Hund();
```

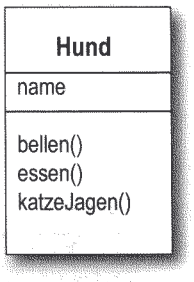



### Spitzen Sie Ihren Bleistift

Was ist der aktuelle Wert von haustiere[2]? _____

Wie sähe der Code aus, der bewirkt, dass haustiere[3] eins der beiden vorhandenen Hund-Objekte referenziert?

_____

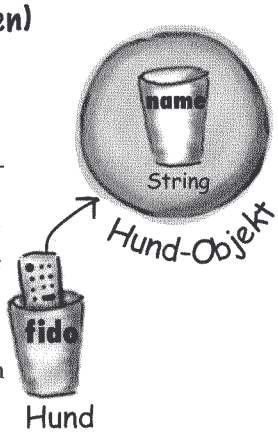



## Ihren Hund steuern

### (mit einer Referenzvariablen)

```
Hund fido = new Hund();
fido.name = "Fido";
```

Wir haben ein Hund-Objekt erzeugt und auf der Referenzvariablen ***fido*** den Punktoperator verwendet, um auf die Variable ***name*** zuzugreifen.*




Wir können die Referenz ***fido*** verwenden, um den Hund zum bellen() oder essen() oder katzeJagen() zu bringen.

```
fido.bellen();
fido.katzeJagen();
```

## Was passiert, wenn der Hund in einem Hund-Array steckt?

Wir wissen, dass wir auf die Instanzvariablen und Methoden des Hundes mit dem Punktoperator zugreifen können. Aber *worauf wenden wir den Punktoperator an?*

Wenn sich der Hund in einem Array befindet, haben wir keinen echten Variablennamen (wie ***fido***). Stattdessen nutzen wir die Array-Notation und drücken den Knopf der Fernsteuerung (den Punktoperator) auf einem Objekt an einem bestimmten Index (einer bestimmten Position) im Array:

```
Hund[] meineHunde = new Hund[3];
meineHunde[0] = new Hund();
meineHunde[0].name = "Fido";
meineHunde[0].bellen();
```

*Ja, uns ist klar, dass das keine Demonstration von Kapselung ist. Aber hier und jetzt wollen wir die Sache einfach halten. Kapselung werden wir uns in Kapitel 4 vornehmen.

### Java kümmert sich um Typen.

**Nachdem Sie ein Array deklariert haben, können Sie nur Dinge hineinstecken, deren Typ dem deklarierten Array-Typ entspricht.**

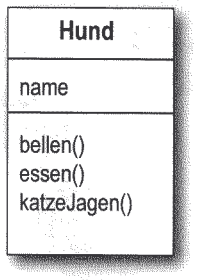
Beispielsweise können Sie keine Katze in ein Hund-Array stecken (es wäre ziemlich scheußlich, wenn jemand davon ausgeht, dass sich in dem Array nur Hunde befinden, alle zum Bellen auffordert und zu seinem Entsetzen feststellen muss, dass da eine Katze herumlungert). Sie können auch kein `double` in ein `int`-Array stecken (denken Sie daran, dass dann etwas verschüttet würde). Sie können allerdings ein `byte` in ein `int`-Array stecken, weil ein `byte` immer in einen Becher mit `int`-Größe passt.

Das wird als implizite Erweiterung bezeichnet. Die Einzelheiten werden wir uns später ansehen. Im Augenblick brauchen Sie sich bloß zu merken, dass der Compiler Ihnen auf Basis des deklarierten Array-Typs verbietet, die falschen Dinge in ein Array zu stecken.

```
class Hund {
  String name;
  public static void main (String[] args) {
    // ein Hund-Objekt erzeugen und darauf
    // zugreifen
    Hund hund1 = new Hund();
    hund1.bellen();
    hund1.name = "Bert";

    // ein Hund-Array erzeugen
    Hund[] meineHunde = new Hund[3];
    // es mit einigen Hunden füllen
    meineHunde[0] = new Hund();
    meineHunde[1] = new Hund();
    meineHunde[2] = hund1;

    // jetzt mit Array-Referenzen aus dem Array
    // auf die Hunde zugreifen
    meineHunde[0].name = "Fred";
    meineHunde[1].name = "Margot";

    // Hmmmm ... wie war noch mal der Name von
    // meineHunde[2]?
    System.out.print("Der Name des letzten Hundes
                      ist ");
    System.out.println(meineHunde[2].name);
    // jetzt das Array in einer Schleife
    // durchlaufen und allen Hunden sagen, dass
    // sie bellen sollen
    int x = 0;
    while(x < meineHunde.length) {
      meineHunde[x].bellen();
      x = x + 1;
    }
  }

  public void bellen() {
    System.out.println(name + " sagt WauWau!");
  }
  public void essen() { }
  public void katzeJagen() { }
}
```

Arrays haben eine Variable length, die die Anzahl der Elemente im Array angibt.

## Ein Hundebeispiel




**Ausgabe**

```
Datei Bearbeiten Fenster Hilfe Heul
%java Hund
null sagt Wauwau!
Der Name des letzten Hundes ist Bert
Fred sagt WauWau!
Margot sagt WauWau!
Bert sagt WauWau!
```

### Punkt für Punkt

- Variablen kommen in zwei Varianten vor: als elementare Typen und als Referenztypen.
- Variablen müssen immer mit einem Namen und einem Typ deklariert werden.
- Der Wert einer elementaren Variablen sind die Bits, die den Wert (5, 'a', true, 3.1416 usw.) repräsentieren.
- Der Wert einer Referenzvariablen sind die Bits, die ein Mittel repräsentieren, an ein Objekt auf dem Heap heranzukommen.
- Eine Referenzvariable ist wie eine Fernsteuerung. Den Punktoperator auf einer Referenzvariable zu verwenden ist, als drücke man den Knopf einer Fernsteuerung, um auf eine Methode oder eine Instanzvariable zuzugreifen.
- Eine Referenzvariable hat den Wert `null`, wenn sie kein Objekt referenziert.
- Ein Array ist immer ein Objekt, selbst wenn deklariert wurde, dass das Array elementare Werte enthält. So etwas wie ein elementares Array gibt es nicht, nur Arrays, die elementare Werte *enthalten*.

# Spielen Sie JVM

Die Java-Dateien auf dieser Seite stellen vollständige Quelldateien dar. Ihr Job ist es, JVM zu spielen und festzustellen, ob sich diese Dateien verarbeiten lassen. Falls nein – wie könnten Sie das in Ordnung bringen?

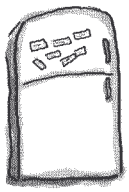
## A

```
class Buch {
  String titel;
  String autor;
}

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


    Buch [] meineBücher = new Buch[3];
    int x = 0;
    meineBücher[0].titel = "Die Früchte von Java";
    meineBücher[1].titel = "Der Java-Gatsby";
    meineBücher[2].titel = "Das Java-Kochbuch";
    meineBücher[0].autor = "richard";
    meineBücher[1].autor = "susanne";
    meineBücher[2].autor = "ian";


    while (x < 3) {
      System.out.print(meineBücher[x].titel);
      System.out.print(" von ");
      System.out.println(meineBücher[x].autor);
      x = x + 1;
    }
  }
}
```

## B

```
class Hobbits {

  String name;

  public static void main(String [] args) {

    Hobbits [] h = new Hobbits[3];
    int z = 0;

    while (z < 4) {
      z = z + 1;
      h[z] = new Hobbits();
      h[z].name = "bilbo";
      if (z == 1) {
        h[z].name = "frodo";
      }
      if (z == 2) {
        h[z].name = "sam";
      }
      System.out.print(h[z].name + " ist ein ");
      System.out.println("guter Hobbit-Name");
    }
  }
}
```

# Code-Magneten



Ein funktionierendes Java-Programm ist vollkommen durcheinander geraten. 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 hinzu, wie Sie benötigen!

```
int y = 0;
```

```
ref = index[y];
```

```
inseln[0] = "Bermudas";
inseln[1] = "Fidschi";
inseln[2] = "Azoren";
inseln[3] = "Kuba";
```

```
int ref;
while (y < 4) {
```

```
System.out.println(inseln[ref]);
```

```
index[0] = 1;
index[1] = 3;
index[2] = 0;
index[3] = 2;
```

```
String [] inseln = new String[4];
```

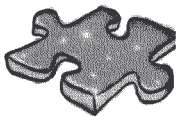
```
System.out.print("Insel = ");
```

```
int [] index = new int[4];
```

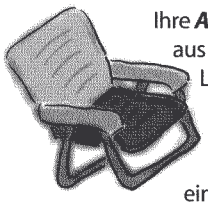
```
y = y + 1;
```

```
Datei Bearbeiten Fenster Hilfe Bikini

% java TestArrays
Insel = Fidschi
Insel = Kuba
Insel = Bermudas
Insel = Azoren
```

```
class TestArrays {

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



## Pool-Puzzle



Ihre ***Aufgabe*** ist es, die Codeschnipsel aus dem Pool zu nehmen und sie in die Lücken im Code einzusetzen. Jeder Schnipsel darf **mehrmals** verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Ihr ***Ziel*** ist es, eine Klasse zu erstellen, die sich kompilieren und ausführen lässt und die hier gezeigte Ausgabe erzeugt.

### Ausgabe

```
Datei Bearbeiten Fenster Hilfe Bermuda
%java Dreieck
Dreieck 0, Fläche = 4.0
Dreieck 1, Fläche = 10.0
Dreieck 2, Fläche = 18.0
Dreieck 3, Fläche = ____
y = _____________________
```

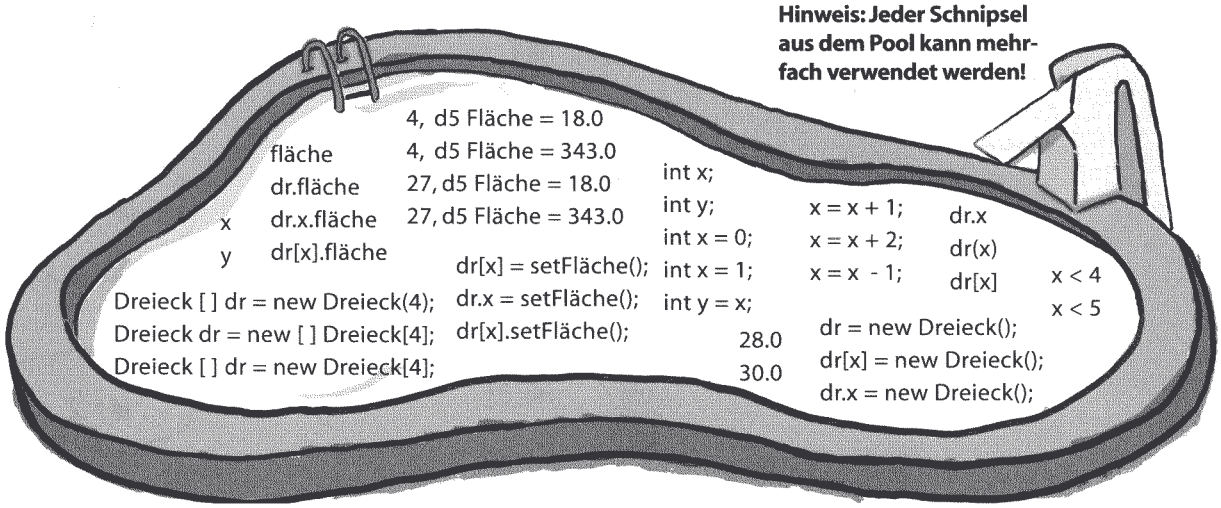
### Bonus-Frage!

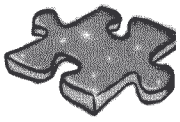
Sie erhalten Bonuspunkte, wenn Sie Schnipsel aus dem Pool verwenden können, um (oben) die fehlende Ausgabe aufzufüllen.

(Manchmal verwenden wir keine separate Testklasse, weil wir versuchen, Platz auf der Seite zu sparen.)

```
class Dreieck {
  double fläche;
  int höhe;
  int breite;
  public static void main(String [] args) {

    ___________

    ___________________

    while ( __________ ) {

      ___________________________

      ________.höhe = (x + 1) * 2;

      ________.breite= x + 4;

      ___________________________

      System.out.print("Dreieck "+x+", Fläche");

      System.out.println(" = " + ________.fläche);

      ________________

    }

    _____________

    x = 27;

    Dreieck d5 = dr[2];

    dr[2].fläche = 343;

    System.out.print("y = " + y);

    System.out.println(", d5 Fläche = "+ d5.fläche);

  }

  void setFläche() {

    _____________ = (höhe * breite) / 2;

  }

}
```

**Hinweis: Jeder Schnipsel aus dem Pool kann mehrfach verwendet werden!**





# Heap-Chaos

Rechts sehen Sie ein kurzes Java-Programm. Wenn »// mach was« erreicht ist, wurden bereits ein paar Objekte und ein paar Referenzvariablen erzeugt. Ihre Aufgabe ist es zu ermitteln, welche der Referenzvariablen welches Objekt referenziert. Nicht alle Referenzvariablen werden verwendet, und es kann sein, dass einige Objekte mehr als einmal referenziert werden. Ziehen Sie Linien, die die Referenzvariablen mit den passenden Objekten verbinden.

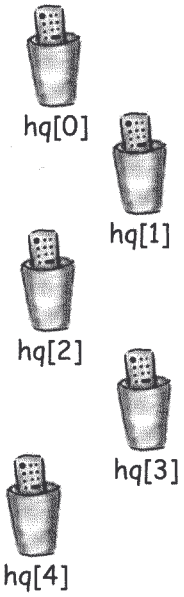
***Tipp:*** Wenn Sie nicht gerade schlauer sind als wir, müssen Sie wahrscheinlich Diagramme wie die auf Seite 57 und 58 dieses Kapitels zeichnen. Benutzen Sie einen Bleistift. Dann können Sie die Referenzverknüpfungen einzeichnen und wieder löschen (die Pfeile zeigen von einer Referenz-Fernsteuerung auf ein Objekt).

```
class HeapQuiz {
  int id = 0;
  public static void main(String [] args) {
    int x = 0;
    HeapQuiz [ ] hq = new HeapQuiz[5];
    while ( x < 3 ) {
      hq[x] = new HeapQuiz();
      hq[x].id = x;
      x = x + 1;
    }
    hq[3] = hq[1];
    hq[4] = hq[1];
    hq[3] = null;
    hq[4] = hq[0];
    hq[0] = hq[3];
    hq[3] = hq[2];
    hq[2] = hq[0];
    // mach was
  }
}
```

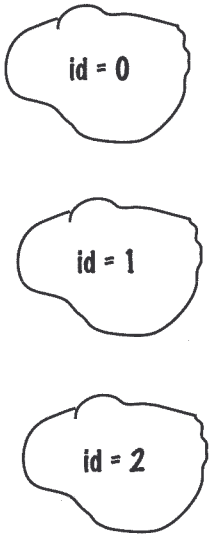
Ordnen Sie den Referenzvariablen das passende Objekt zu.

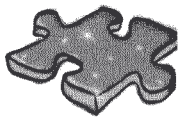
Eventuell müssen Sie nicht alle Referenzen verwenden.

**Referenzvariablen:**




**HeapQuiz -Objekte:**





## Der Fall der geraubten Referenzen

Es war eine dunkle und stürmische Nacht. Carla schlenderte ins Programmierer-Loch, als gehörte ihr der Laden. Sie wusste, dass all die Programmierer immer noch ihrem harten Job nachgingen. Und sie suchte Hilfe. Sie brauchte eine neue Methode für eine enorm wichtige Klasse, die in das neue, streng geheime, Java-fähige Handy ihres Kunden geladen werden sollte. Der Heap-Platz im Speicher des Handys war so eng geschnitten wie Carlas Top, und alle wussten es. Das üblicherweise lärmige Gegröle im Loch ebbte ab, als Carla sich ihren Weg zur Tafel bahnte. Sie skizzierte einen kurzen Überblick über Funktionalität der neuen Methode und ließ ihren Blick durch den Raum schweifen. »Also Jungs, die Stunde hat geschlagen«, schnurrte sie.

»Wer von euch die speichereffizienteste Version dieser Methode schreibt, kann morgen mit mir zur Launch-Party meines Kunden nach Rio fliegen ... und mir helfen, die neue Software zu installieren.«

Am nächsten Morgen glitt Carla in ihrem kurzen Strandkleid ins Loch. »Boys«, lächelte sie, »das Flugzeug geht in ein paar Stunden. Zeigt mir, was ihr vorzuweisen habt!« Boris war der Erste. Als er begann, seinen Entwurf auf die Tafel zu kritzeln, zischte Carla: »Komm zur Sache, Boris. Zeig mir, wie du das mit der Aktualisierung der Kontakt-Objekte geregelt hast.« Hastig warf Boris ein Codefragment auf die Tafel:

```
Kontakt[] k = new Kontakt[10];
while ( x < 10 ) { // 10 Kontakt-Objekte machen
  k[x] = new Kontakt();
  x = x + 1;
}
// mit k kompliziertes Zeug zur Aktualisierung
// der Kontaktliste vornehmen
```

»Ja, Carla, ich weiß, dass der Speicherplatz knapp ist. Aber deine Spec sagt, dass es möglich sein soll, auf individuelle Kontaktinformationen für alle 10 zulässigen Kontakte zuzugreifen. Das war das beste Rezept, das ich zusammenkochen konnte«, sagte Boris. Kurt war als Nächster an der Reihe. Innerlich sah er sich bereits bei einem Kokosnuss-Cocktail mit Carla. »Boris,« sagte er, »meinst du nicht, dass deine Lösung etwas mickrig ist?«, und grinste: »Sieh dir das an, Püppchen!«:

```
Kontakt refK;
while ( x < 10 ) { // 10 Kontakt-Objekte machen
  refK = new Kontakt();
  x = x + 1;
}
// mit refK kompliziertes Zeug zur Aktualisierung
// der Kontaktliste vornehmen
```

»Ich habe Speicherplatz für einen ganzen Haufen von Referenzvariablen gespart, Bobbele. Pack deine Badehose besser wieder ein«, feixte Kurt. »Nicht so schnell, Kurti!«, sagte Carla, »du hast zwar etwas Speicherplatz gespart, aber Boris ist der, der mit mir kommt.«

***Warum hat sich Carla für Boris' Methode entschieden, obwohl Kurts weniger Speicherplatz benötigt?***

# Lösungen zu den Übungen

## Code-Magneten:

```
class TestArrays {
  public static void main(String [] args) {
    int [] index = new int[4];
    index[0] = 1;
    index[1] = 3;
    index[2] = 0;
    index[3] = 2;
    String [] inseln = new String[4];
    inseln[0] = "Bermudas";
    inseln[1] = "Fidschi";
    inseln[2] = "Azoren";
    inseln[3] = "Kuba";
    int y = 0;
    int ref;
    while (y < 4) {
      ref = index[y];
      System.out.print("Insel = ");
      System.out.println(inseln[ref]);
      y = y + 1;
    }
  }
}
```

```
Datei Bearbeiten Fenster Hilfe Bikini

% java TestArrays
Insel = Fidschi
Insel = Kuba
Insel = Bermudas
Insel = Azoren
```

**A**

```
class Buch {
  String titel;
  String autor;
}
class BücherTestlauf {
  public static void main(String [] args) {
    Buch [] meineBücher = new Buch[3];
    int x = 0;
    meineBücher[0] = new Buch();
    meineBücher[1] = new Buch();
    meineBücher[2] = new Buch();
    meineBücher[0].titel = "Die Früchte von Java";
    meineBücher[1].titel = "Der Java-Gatsby";
    meineBücher[2].titel = "Das Java-Kochbuch";
    meineBücher[0].autor = "richard";
    meineBücher[1].autor = "susanne";
    meineBücher[2].autor = "ian";
    while (x < 3) {
      System.out.print(meineBücher[x].titel);
      System.out.print(" von ");
      System.out.println(meineBücher[x].autor);
      x = x + 1;
    }
  }
}
```

Denken Sie daran: Wir müssen erst die Buch-Objekte erstellen!

---

**B**

```
class Hobbits {
  String name;
  public static void main(String [] args) {
    Hobbits [] h = new Hobbits[3];
    int z = -1;
    while (z < 2) {
      z = z + 1;
      h[z] = new Hobbits();
      h[z].name = "bilbo";
      if (z == 1) {
        h[z].name = "frodo";
      }
      if (z == 2) {
        h[z].name = "sam";
      }
      System.out.print(h[z].name + " ist ein ");
      System.out.println("guter Hobbit-Name");
    }
  }
}
```

Denken Sie daran: Arrays beginnen bei Element 0!

# Puzzle-Lösungen

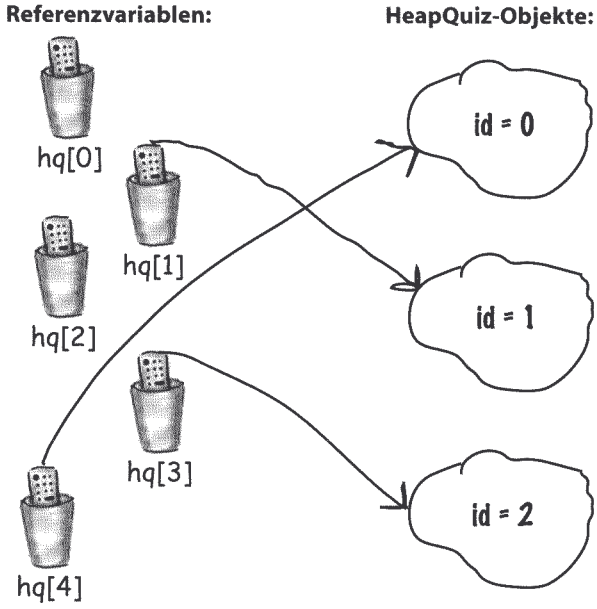
```
class Dreieck {
  double fläche;
  int höhe;
  int breite;
  public static void main(String [] args) {
    int x = 0;
    Dreieck[ ] dr = new Dreieck[4];
    while ( x < 4 ) {
      dr[x] = new Dreieck();
      dr[x].höhe = (x + 1) * 2;
      dr[x].breite = x + 4;
      dr[x].setFläche();
      System.out.print("Dreieck "+x+", Fläche");
      System.out.println(" = " + dr[x].fläche);
      x = x + 1;
    }
    int y = x;
    x = 27;
    Dreieck d5 = dr[2];
    dr[2].fläche = 343;
    System.out.print("y = " + y);
    System.out.println(", d5 Fläche = "+ d5.fläche);
  }
  void setFläche() {
    fläche = (höhe * breite) / 2;
  }
}
```

```
Datei Bearbeiten Fenster Hilfe Bermuda
%java Dreieck
Dreieck 0, Fläche = 4.0
Dreieck 1, Fläche = 10.0
Dreieck 2, Fläche = 18.0
Dreieck 3, Fläche = 28.0
y = 4, d5 Fläche = 343
```

## Der Fall der geraubten Referenzen

Carla sah, dass Kurts Methode einen gewichtigen Fehler hatte. Es ist richtig, dass er nicht so viele Referenzvariablen benötigte wie Boris. Aber es gab keine Möglichkeit, auf ein anderes als das letzte Kontakt-Objekt zuzugreifen, das von dieser Methode erzeugt wurde. Bei jedem Trip durch die Schleife wies er der einen Referenzvariablen ein neues Objekt zu. Das zuvor referenzierte Objekt wurde also auf dem Heap verlassen – es wurde *unerreichbar*. Ohne Zugriff auf neun von zehn Objekten war Kurts Methode vollkommen nutzlos.

(Das Programm war ein gewaltiger Erfolg, und der Kunde spendierte Carla und Boris eine zusätzliche Woche an der Copa Cabana. Gern würden wir Sie damit locken, dass Sie so etwas auch bekommen, wenn Sie dieses Buch durchgearbeitet haben, aber ...)

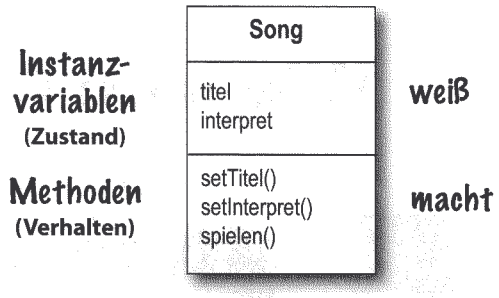


# Wie sich Objekte verhalten

**Zustände haben Auswirkungen auf Verhalten, und Verhalten haben Auswirkungen auf Zustände.** Wir wissen, dass Objekte **Zustände** und **Verhalten** haben, die von **Instanzvariablen** und **Methoden** repräsentiert werden. Aber bisher haben wir uns noch nicht angesehen, in welcher **Beziehung** Zustände und Verhalten zueinander **stehen**. Wir wissen bereits, dass jede Instanz einer Klasse (jedes Objekt eines bestimmten Typs) eigene Werte für seine Instanzvariablen haben kann. Hund A kann den **name**n »Fido« und das **gewicht** 35 Kilo haben. Hund B heißt »Killer« und wiegt 5 Kilo. Und wenn die Klasse Hund eine Methode machKrach() besitzt, meinen Sie doch auch, dass ein 35-Kilo-Hund etwas tiefer bellt als ein 5-Kilo-Knäuel, oder? (Vorausgesetzt, dieses Kläffgeräusch kann als ***Bellen*** bezeichnet werden.) Glücklicherweise ist das genau der springende Punkt bei Objekten – sie haben ein **Verhalten**, das auf ihre **Zustände** wirkt. Anders gesagt, ***Methoden nutzen die Werte von Instanzvariablen*** – beispielsweise »wenn Hund weniger wiegt als 7 Kilo, mach Kläffgeräusch« oder »erhöhe Gewicht um 3 Kilo«. ***Ändern wir mal ein paar Zustände.***

# Erinnern Sie sich: Eine Klasse beschreibt, was ein Objekt weiß und was ein Objekt macht

**Eine Klasse ist eine Blaupause für ein Objekt.** Wenn Sie eine Klasse schreiben, beschreiben Sie, wie die JVM ein Objekt dieses Typs machen soll. Sie wissen auch, dass jedes Objekt dieses Typs andere *Instanzvariablenwerte* haben kann. Aber was ist mit den Methoden?




## Kann jedes Objekt dieses Typs ein anderes Methodenverhalten haben?

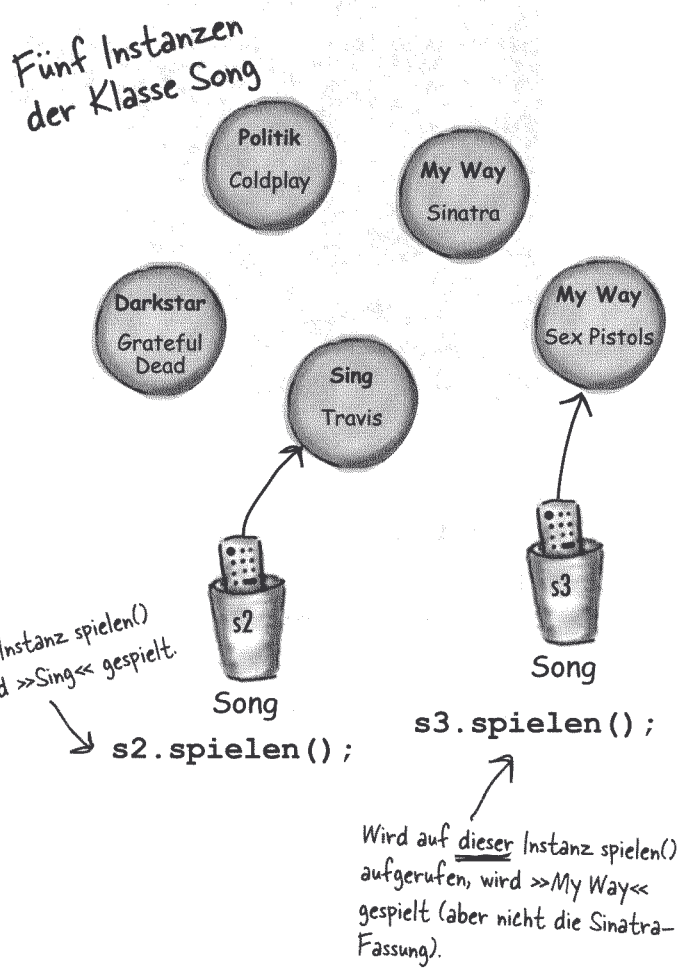
Hm ... *so ähnlich.**

Jede Instanz einer bestimmten Klasse hat die gleichen Methoden, aber in Abhängigkeit von den Werten der Instanzvariablen können sich die Methoden anders *verhalten.*

Die Klasse Song hat zwei Instanzvariablen, *titel* und *interpret.* Die Methode spielen() spielt einen Song, aber die Instanz, auf der Sie spielen() aufrufen, spielt den Song ab, der durch die *titel*-Instanzvariable dieser Instanz repräsentiert wird. Wenn Sie die Methode spielen() auf der einen Instanz aufrufen, hören Sie deswegen das Lied »Politik«. Eine andere Instanz hingegen spielt »Darkstar«. Aber der Methodencode bleibt derselbe.

```
void spielen() {
    musikGerät.spieleSong(titel);
}
```

```
Song s2 = new Song();
s2.setInterpret("Travis");
s2.setTitel("Sing");

Song s3 = new Song();
s3.setInterpret("Sex Pistols");
s3.setTitel("My Way");
```

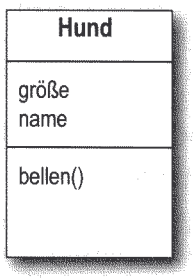



*Super, mal wieder eine dieser unglaublich eindeutigen Antworten!

# Die Größe beeinflusst das Bellen

Das Bellen eines kleinen Hundes unterscheidet sich vom Bellen eines großen Hundes.

Die Klasse Hund besitzt eine Instanzvariable *größe*, die die Methode *bellen()* verwendet, um zu entscheiden, welche Art von Bellgeräusch gemacht werden soll.

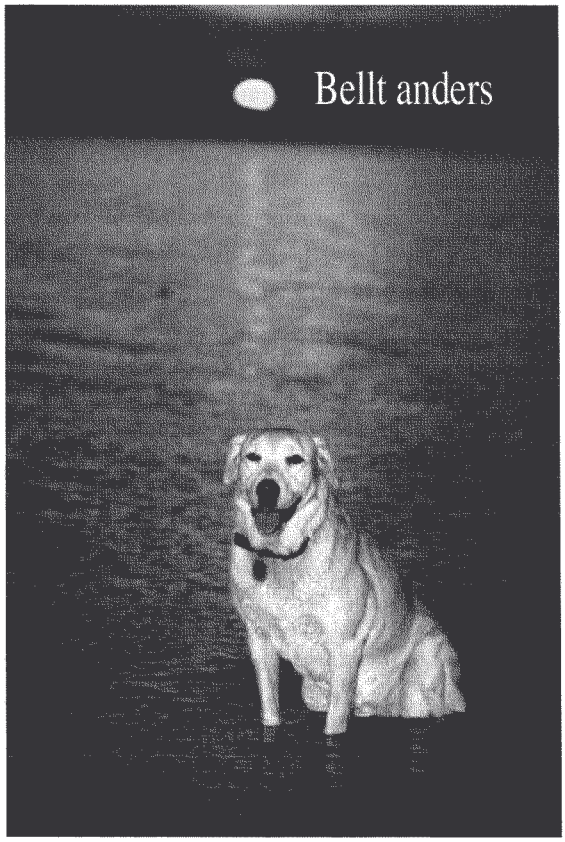



```
class Hund {
  int größe;
  String name;

  void bellen() {
    if (größe > 60) {
       System.out.println(»Wuff! Wuff!");
    } else if (größe > 14) {
       System.out.println(»Wau! Wau!");
    } else {
       System.out.println(»Jip! Jip!");
    }
  }
}
```

---

```
class HundTestlauf {

   public static void main (String[] args) {
     Hund eins = new Hund();
     eins.größe = 70;
     Hund zwei = new Hund();
     zwei.größe = 8;
     Hund drei = new Hund();
     drei.größe = 35;

     eins.bellen();
     zwei.bellen();
     drei.bellen();
   }
}
```

```
Datei Bearbeiten Fenster Hilfe SpielToterMann
%java HundeTestlauf
Wuff! Wuff!
Jip! Jip!
Wau!  Wau!
```



# Sie können einer Methode Informationen schicken

Wie von einer Programmiersprache zu erwarten, können Sie Ihren Methoden Werte übergeben. Beispielsweise möchten Sie einem Hund-Objekt vielleicht sagen, wie gebellt werden soll, indem Sie folgenden Aufruf verwenden:

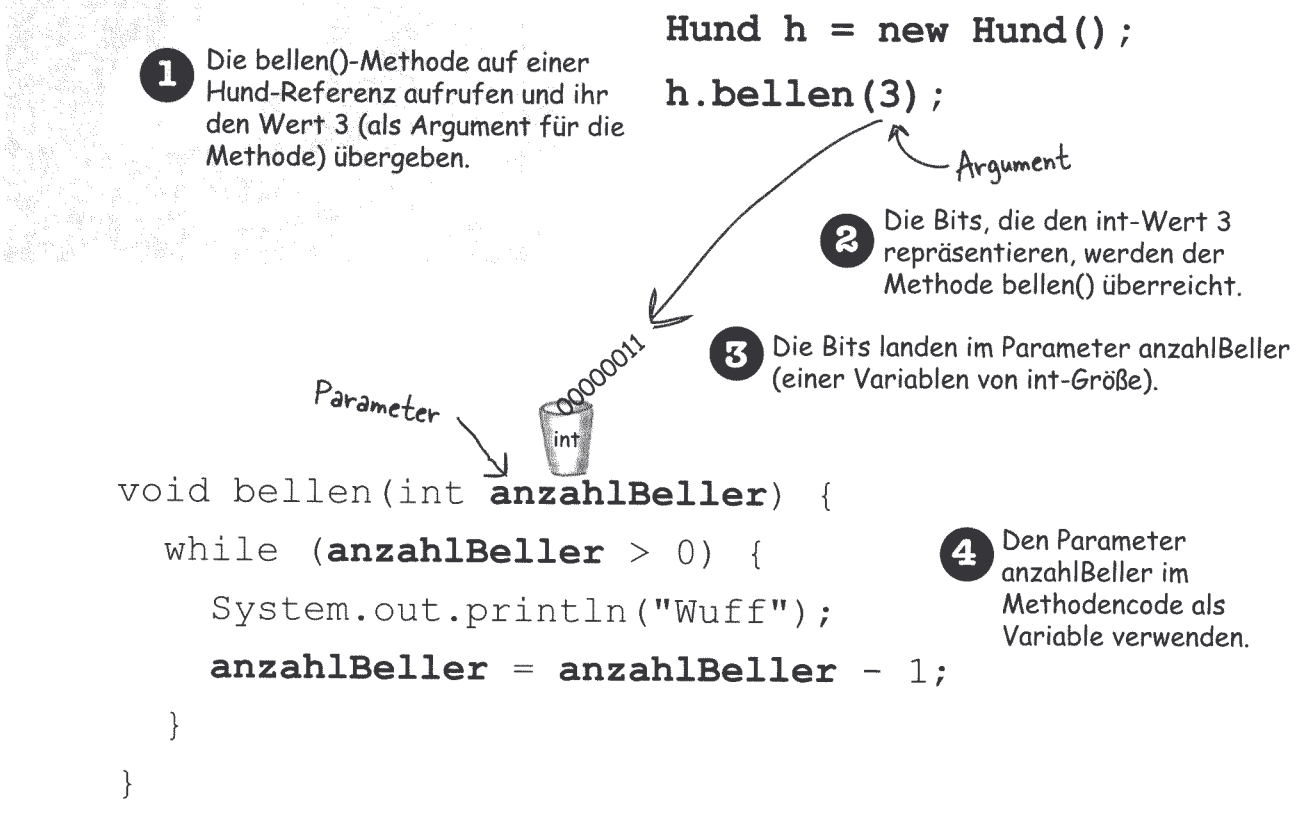
```
h.bellen(3);
```

Abhängig von Ihrem Programmierhintergrund und Ihren persönlichen Vorlieben verwenden *Sie* wahrscheinlich den Begriff *Argument* oder vielleicht *Parameter* für die Werte, die einer Methode übergeben werden. Obwohl es in der Computerwissenschaft formale Unterschiede zwischen diesen Begriffen *gibt*, die von Menschen gemacht werden, die Laborkittel tragen und dieses Buch mit Sicherheit nicht lesen werden, kümmern wir uns hier lieber um die wirklich großen Fische. *Sie* können diese Werte also nennen, wie es Ihnen passt (Argumente, Krapfen, Fussel oder so), wir machen das folgendermaßen:

**Eine Methode nutzt Parameter. Ein Aufrufer übergibt Argumente.**

Argumente sind die Dinge, die Sie den Methoden übergeben. Ein ***Argument*** (ein Wert wie 2, »Foo« oder eine Referenz auf einen Hund) stürzt kopfüber in einen ... warten Sie ... ***Parameter***. Und ein Parameter ist nichts anderes als eine lokale Variable. Eine Variable mit einem Typ und einem Namen, die innerhalb des Methodenrumpfs verwendet werden kann.

Aber jetzt kommt der wichtige Teil: **Wenn eine Methode einen Parameter erwartet, dann *müssen* Sie ihr etwas übergeben.** Und dieses Etwas muss ein Wert mit einem geeigneten Typ sein.



# Sie können etwas von Methoden *zurückbekommen*

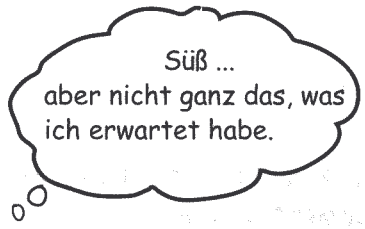
Methoden können Werte zurückliefern. Jede Methode wird mit einem Rückgabetyp deklariert. Aber unsere Methoden haben wir bisher mit dem Rückgabetyp **void** gemacht, was bedeutet, dass sie nichts zurückliefern.

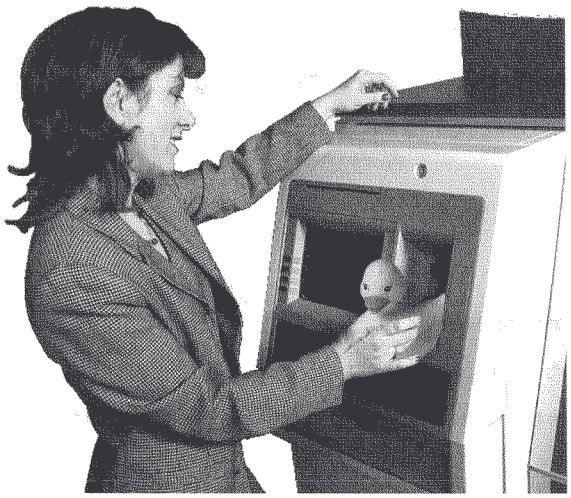
```
void los() {
}
```

Aber wir können deklarieren, dass eine Methode dem Aufrufer einen bestimmten Typ Wert liefert:

```
int gibGeheimnis() {
  return 42;
}
```

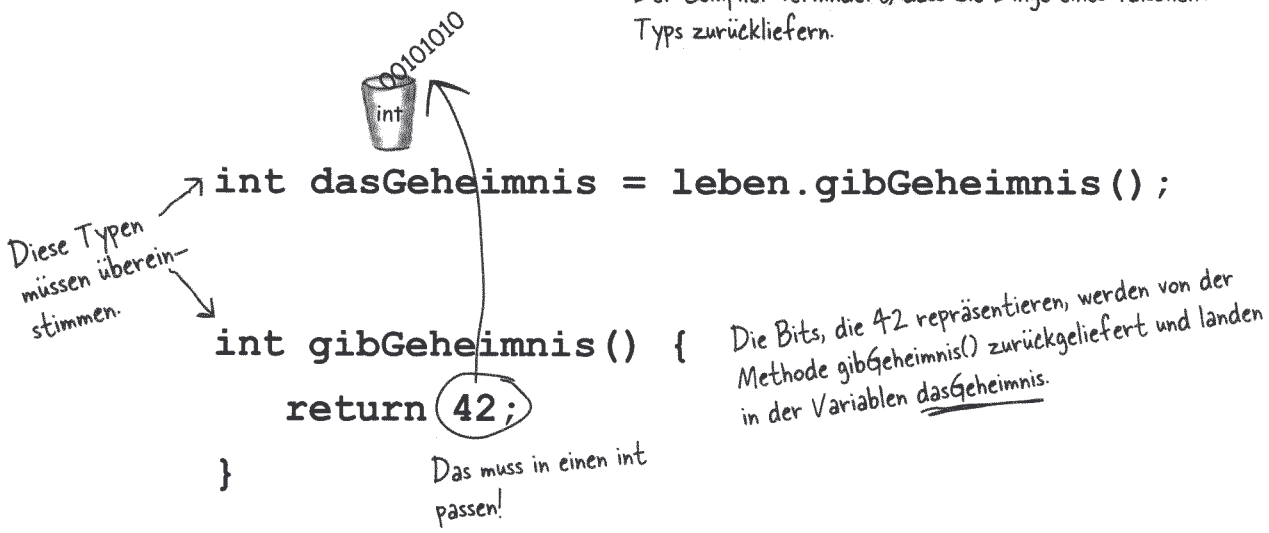
Wenn Sie deklarieren, dass eine Methode einen Wert zurückliefert, *müssen* Sie einen Wert des deklarierten Typs zurückliefern! (Oder einen Wert, der mit dem deklarierten Typ *kompatibel* ist. Dem werden wir uns ausführlicher widmen, wenn wir in den Kapiteln 7 und 8 über Polymorphie reden.)



Der Compiler verhindert, dass Sie Dinge eines falschen Typs zurückliefern.

## Was immer Sie zu liefern versprechen, Sie sollten es dann auch tun!



# Sie können einer Methode mehr als eine Sache schicken

Methoden können mehrere Parameter haben. Trennen Sie die Parameter durch Kommata, wenn Sie sie deklarieren, und trennen Sie die Argumente durch Kommata, wenn Sie sie übergeben. Das Wichtigste aber ist, dass Sie, wenn eine Methode Parameter hat, Argumente des richtigen Typs in der richtigen Reihenfolge übergeben *müssen*.

## Eine Methode mit zwei Parametern aufrufen und ihr zwei Argumente übergeben.

```
void los() {
   TestZeug t = new TestZeug();
   t.nimmZwei(12, 34);
}

void nimmZwei(int x, int y) {
   int z = x + y;
   System.out.println("Die Summe ist " + z);
}
```

Die Argumente, die Sie übergeben, kommen in genau der Reihenfolge an, in der Sie sie übergeben haben. Das erste Argument landet im ersten Parameter, das zweite Argument im zweiten Parameter und so weiter.

## Sie können einer Methode Variablen übergeben, wenn die Variablentypen den Parametertypen entsprechen.

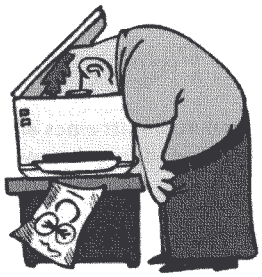
```
void los() {
   int foo = 7;
   int bar = 3;
   t.nimmZwei(foo, bar);
}

void nimmZwei(int x, int y) {
   int z = x + y;
   System.out.println("Die Summe ist " + z);
}
```

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

Was ist der Wert von z? Das Ergebnis entspricht dem, das Sie erhalten hätten, wenn Sie foo und bar zu dem Zeitpunkt addiert hätten, als Sie sie der Methode nimmZwei übergaben.

# Java ist Pass-by-Value.

# Das bedeutet Pass-by-Copy.

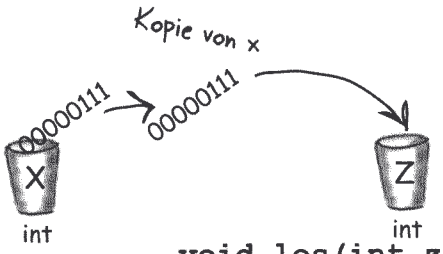


```
int x = 7;
```

**1** Eine int-Variable deklarieren und ihr den Wert »7« zuweisen. Das Bitmuster für 7 kommt in die Variable namens x.

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

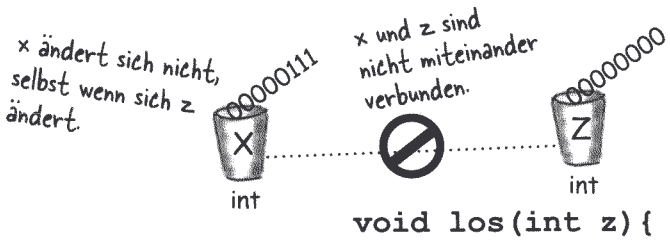
**2** Eine Methode mit einem int-Parameter namens z deklarieren.




```
foo.los(x);
```

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

**3** Die Methode los() aufrufen und ihr dabei die Variable x als Argument übergeben. Die Bits in x werden kopiert, und die Kopie landet in z.




```
void los(int z){
    z = 0;
}
```

**4** Den Wert von z innerhalb der Methode ändern. Der Wert von x ändert sich nicht! Das an den Parameter z übergebene Argument war nur eine Kopie von x.

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

## Es gibt keine Dummen Fragen

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

A: Dazu werden Sie in späteren Kapiteln mehr erfahren. Aber eigentlich *wissen* Sie die Antwort schon. Java übergibt *alles* by value, also als Wert. ***Alles***. Aber ... *Wert*, das sind *Bits innerhalb der Variablen*. Und Sie erinnern sich, dass man keine Objekte in Variablen steckt. Die Variable ist eine Fernsteuerung – *eine Referenz auf ein Objekt*. Wenn Sie eine Referenz auf ein Objekt in eine Methode übergeben, übergeben Sie also eine *Kopie der Fernsteuerung*. Halten Sie durch – dazu haben wir noch eine ganze Menge zu sagen.

**F: Kann eine Methode mehrere Rückgabewerte deklarieren? Oder gibt es eine Möglichkeit, mehr als einen Wert zurückzuliefern?**

A: So ähnlich. Eine Methode kann nur einen Rückgabewert deklarieren. ABER ... wenn Sie, sagen wir, drei int-Werte zurückliefern wollen, kann der deklarierte Rückgabewert ein int-*Array* sein. Stecken Sie die ints in das Array und liefern Sie es zurück. Es ist etwas komplizierter, wenn Sie mehrere Werte mit unterschiedlichen Typen zurückliefern möchten. Das sehen wir uns später an, wenn wir über ArrayList reden.

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

A: Sie dürfen alles zurückliefern, das *implizit* in den deklarierten Typ umgewandelt werden kann. Sie könnten also einen byte-Wert zurückliefern, wo ein int-Wert erwartet wird. Dem Aufrufer ist das egal, weil der byte-Wert problemlos in die int-Variable passt, die der Aufrufer verwendet, um das Ergebnis festzuhalten. Sie müssen aber eine *explizite* Umwandlung vornehmen, wenn der deklarierte Typ *kleiner* ist als der Typ des Werts, den Sie zurückliefern möchten.

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

A: Java zwingt Sie nicht, einen Rückgabewert zu berücksichtigen. Manchmal möchten Sie vielleicht eine Methode aufrufen, deren Rückgabetyp nicht void ist, obwohl Sie sich für den Rückgabewert nicht interessieren. In einem solchen Fall rufen Sie die Methode wegen der Arbeit auf, die *innerhalb* der Methode erfolgt, nicht wegen des Ergebnisses, das die Methode *zurückliefert*. In Java müssen Sie den Rückgabewert nicht zuweisen oder verwenden.

## Erinnerung: Java kümmert sich um Typen!

**Sie können keine Giraffe zurückliefern, wenn der Rückgabetyp als Kaninchen deklariert ist. Das Gleiche gilt für Parameter. Einer Methode, die ein Kaninchen erwartet, können Sie keine Giraffe übergeben.**

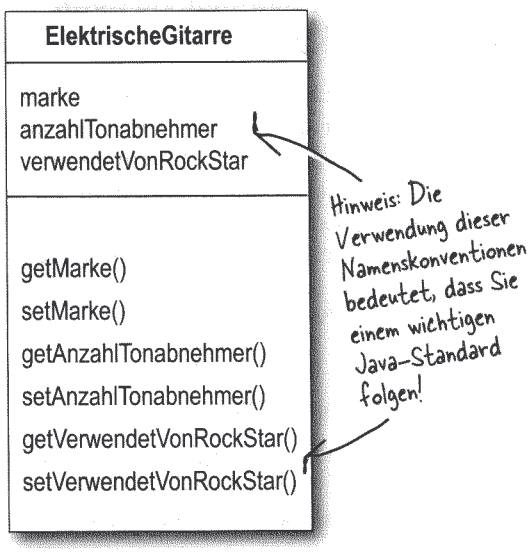
## Punkt für Punkt

- Klassen definieren, was ein Objekt weiß und was ein Objekt macht.
- Dinge, die ein Objekt weiß, sind seine **Instanzvariablen** (Zustand).
- Dinge, die ein Objekt macht, sind seine **Methoden** (Verhalten).
- Methoden können Instanzvariablen nutzen, damit sich Objekte des gleichen Typs verschieden verhalten.
- Eine Methode kann Parameter haben. Das heißt, dass Sie einer Methode einen oder mehrere Werte übergeben können.
- Die Anzahl und die Typen der Argumente, die Sie übergeben, müssen der Anzahl und den Typen der Parameter entsprechen, die von der Methode deklariert werden.
- Werte, die Sie an Methoden übergeben oder aus Methoden zurückliefern, können implizit zu einem größeren Typ befördert oder explizit in einen kleineren Typ umgewandelt werden.
- Der Wert, den Sie einer Methode als Argument übergeben, kann ein literaler Wert (2, 'c' usw.) oder eine Variable des deklarierten Parametertyps sein (beispielsweise *x*, wobei *x* eine int-Variable ist). (Es gibt noch andere Dinge, die Sie als Argumente übergeben können, aber so weit sind wir noch nicht.)
- Eine Methode *muss* einen Rückgabetyp deklarieren. Der Rückgabetyp void bedeutet, dass die Methode nichts zurückliefert.
- Wenn eine Methode einen anderen Rückgabetyp als void deklariert, *muss* sie einen Wert zurückliefern, der mit dem deklarierten Rückgabetyp kompatibel ist.

# Coole Sachen, die Sie mit Parametern und Rückgabetypen anstellen können

Jetzt, da wir gesehen haben, wie Parameter und Rückgabetypen funktionieren, ist es Zeit, sie zum Einsatz zu bringen: **Getter** und **Setter**. Wenn Sie das ganz formal ausdrücken möchten, ziehen Sie es vielleicht vor, von *Zugriffsmethoden* und *Veränderungsmethoden* zu sprechen. Aber das hieße Silben verschwenden, die eigentlich vollkommen unnötig sind. Außerdem entsprechen Getter und Setter den Java-Namenskonventionen. Deswegen werden wir diese Namen verwenden.

Getter und Setter ermöglichen Ihnen, *Dinge festzulegen und abzurufen*, in der Regel Werte von Instanzvariablen. Der einzige Lebenszweck eines Getter ist es, als Rückgabewert den Wert dessen zurückzuliefern, das dieser bestimmte Getter abrufen soll. Und jetzt ist es sicher nicht mehr überraschend, dass ein Setter für die Möglichkeit lebt und atmet, einen Argumentwert zu nehmen und zu verwenden, um den Wert einer Instanzvariablen *zu setzen*.




```
public class ElektrischeGitarre {

  String marke;
  int anzahlTonabnehmer;
  boolean verwendetVonRockStar;

  String getMarke() {
    return marke;
  }

  void setMarke(String eineMarke) {
    marke = eineMarke;
  }

  int getAnzahlTonabnehmer() {
    return anzahlTonabnehmer;
  }

  void setAnzahlTonabnehmer(int zahl) {
    anzahlTonabnehmer = zahl;
  }

  boolean getVerwendetVonRockStar() {
    return verwendetVonRockStar;
  }

  void setVerwendetVonRockStar(boolean jaOderNein) {
    verwendetVonRockStar = jaOderNein;
  }
}
```

# Kapselung

## Tun Sie es oder nehmen Sie Hohn und Spott in Kauf.

Bis zu diesem höchst wichtigen Moment haben wir einen der schlimmsten OO-Fauxpas begangen (und wir reden nicht von einer klitzekleinen Regelverletzung, die man beispielsweise begeht, wenn man zu einer Bottleparty ohne Getränke aufkreuzt). Nein, wir reden von einem FauxPas mit einem großen F und einem großen P.

Unsere peinliche Übertretung?

Unsere Daten zu entblößen!

Hier sitzen wir, summen sorglos vor uns hin und lassen unsere Daten einfach so herumstehen, dass *jeder* sie sehen und anpacken kann.

Vielleicht haben Sie bereits dieses seltsam unbehagliche Gefühl gespürt, das auftritt, wenn man seine Instanzvariablen entblößt. Instanzvariablen entblößen heißt, dass man auf sie über den Punktoperator zugreifen kann, zum Beispiel:

```
dieKatze.größe = 27;
```

Denken Sie mal genau über die Idee nach, unsere Fernsteuerung zu nutzen, um die größe-Instanzvariable des Katze-Objekts direkt zu ändern. In den Händen der falschen Personen ist eine Referenzvariable (Fernsteuerung) eine ziemlich gefährliche Waffe. Denn was verhindert dann:

```
dieKatze.größe = 0;
```

Hilfe! Wir dürfen nicht zulassen, dass das passiert!

Das wäre eine üble Geschichte! Wir müssen Setter-Methoden für alle Instanzvariablen einrichten und eine Möglichkeit finden, die Setter aufzurufen, anstatt direkt auf die Daten zuzugreifen.

Indem wir erzwingen, dass alle eine Setter-Methode aufrufen, können wir die Katze vor inakzeptablen Größenänderungen schützen.

```
public void setGröße(int g) {
    if (g > 9) {
        größe = g;
    }
}
```

Bauen wir Tests ein, um ein Minimum für die Katzengröße zu garantieren!

## Daten verbergen

Ja, *so* leicht kommt man von einer Implementierung, die geradezu um korrupte Daten bettelt, zu einer, die Ihre Daten schützt *und* Ihnen das Recht bewahrt, Ihre Implementierung später zu ändern. Okay, wie genau *verbirgt* man Daten? Mit den Zugriffsmodifizierern (oder Modifiern) **`public`** und **`private`**. **`public`** kennen Sie bereits – wir haben ihn bei allen main()-Methoden verwendet.

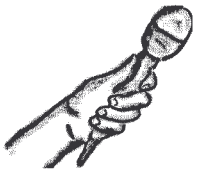
Hier ist eine Faustregel für *Kapselungsanfänger* (natürlich gelten auch hier alle üblichen Garantieeinschränkungen für Faustregeln): Markieren Sie Ihre Instanzvariablen als ***private*** und bieten Sie ***public***-Getter und -Setter zur Steuerung des Zugriffs. Wenn Sie ein breiteres Entwurfs- und Programmierwissen in Java haben, machen Sie diese Sache vielleicht etwas anders, aber für den Augenblick bleiben Sie mit diesem Verfahren auf der sicheren Seite.

**Markieren Sie Instanzvariablen als `private`.**

**Markieren Sie Getter und Setter als `public`.**

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

(an der Kaffeemaschine aufgeschnappt)



## Java im Gespräch

### Interview der Woche: Ein Objekt erklärt, was es mit der Kapselung wirklich auf sich hat.

**Von Kopf bis Fuß:** Was ist denn eigentlich so wichtig an der Kapselung?

**Objekt:** Sie kennen doch diesen Traum: Sie stehen vor 500 Menschen und sollen eine Rede halten … und dann fällt Ihnen plötzlich auf – Sie *sind nackt*!

**Von Kopf bis Fuß:** Ja, den hatten wir auch schon mal. Der ist so ähnlich wie der mit dem Pilates-Gerät und … nein, das lassen wir jetzt lieber. Sie fühlen sich also nackt. Aber mal abgesehen davon, dass man etwas entblößt ist, gibt es irgendwelche Risiken?

**Objekt:** Ob es Risiken gibt? Ob es *irgendwelche* Risiken gibt? [Lacht] Ne, ihr anderen Instanzen, habt ihr das gehört, er fragt danach, *ob es irgendwelche Risiken gibt!!* [kippt vor Lachen vom Stuhl]

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

**Objekt:** Na gut. Ich werde es erklären. Es ist … [bricht erneut in unkontrollierbares Lachen aus]

**Von Kopf bis Fuß:** Brauchen Sie etwas? Wasser?

**Objekt:** Puh! Junge. Nein, mir geht's gut, wirklich. Ich bleib jetzt ernst. Tief atmen. Gut, fahren Sie fort.

**Von Kopf bis Fuß:** Wovor also schützt Sie die Kapselung?

**Objekt:** Kapselung baut um meine Instanzvariablen ein Kraftfeld auf, damit niemand sie auf etwas, sagen wir, *Unpassendes* setzen kann.

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

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

**Von Kopf bis Fuß:** Ich sehe, was Sie meinen. Wie also ermöglicht Ihnen Kapselung, Grenzen zu setzen?

**Objekt:** Indem sie anderen Code zwingt, immer über Setter-Methoden zu gehen. Auf diese Weise kann die Setter-Methode den Parameter validieren und entscheiden, ob er sinnvoll ist. Vielleicht verwirft die Methode den Wert und macht nichts oder löst eine Exception aus (beispielsweise wenn es sich um die Sozialversicherungsnummer bei einer Kreditkartenanwendung handelt) oder rundet den eingesandten Parameter auf den nächsten akzeptablen Wert. Der Punkt ist, das Sie in einer Setter-Methode machen können, was Sie machen möchten. Wenn Ihre Instanzvariablen public sind, können Sie nämlich *rein gar nichts* machen.

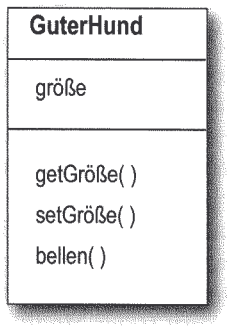
**Von Kopf bis Fuß:** Aber manchmal sieht man Setter-Methoden, die einfach den Wert setzen, ohne irgendetwas zu prüfen. Wenn Sie eine Instanzvariable haben, bei der es keine Grenzen gibt, ist die Setter-Methode dann nicht unnötige Verschwendung? Ein Leistungsminderer?

**Objekt:** Der Punkt bei Settern (und Gettern natürlich auch) ist, ***dass Sie später Ihre Absicht ändern können, ohne dass das den Code anderer zerbricht!*** Stellen Sie sich vor, die halbe Belegschaft Ihres Unternehmens verwendet Ihre öffentlichen Instanzvariablen, und Sie stellen eines Tages plötzlich fest: »Hoppla, da gibt es etwas, das ich bei dem Wert nicht eingeplant habe, ich muss da doch eine Setter-Methode einbauen …« Damit machen Sie den Code aller anderen kaputt. Die coole Sache bei Kapselung ist, dass *Sie Ihre Absichten ändern können.* Und trotzdem wird niemand verletzt. Das, was man an Leistung gewinnt, wenn man Variablen direkt nutzt, ist gering und es selten – *wenn überhaupt irgendwann* – wert, dass man auf Setter verzichtet.

# Die Klasse GuterHund kapseln

Die Instanzvariablen privat machen.

Die Getter- und Setter-Methoden öffentlich machen.




```
class GuterHund {

  private int größe;

  public int getGröße() {
    return größe;
  }

  public void setGröße(int s) {
    größe = s;
  }

  void bellen() {
    if (größe > 60) {
      System.out.println("Wuff! Wuff!");
    } else if (größe > 14) {
      System.out.println("Wau! Wau!");
    } else {
      System.out.println("Jip! Jip!");
    }
  }
}

class GuterHundTestlauf {

  public static void main (String[] args) {
    GuterHund eins = new GuterHund();
    eins.setGröße(70);
    GuterHund zwei = new GuterHund();
    zwei.setGröße(8);
    System.out.println("Hund eins: " + eins.getGröße());
    System.out.println("Hund zwei: " + zwei.getGröße());
    eins.bellen();
    zwei.bellen();
  }
}
```

Auch wenn die Methoden nicht wirklich neue Funktionalitäten hinzufügen, ist das Gute an ihnen, dass Sie später Ihre Meinung ändern und eine Methode sicherer, schneller, besser machen können.

**An jeder Stelle, an der ein bestimmter Wert verwendet werden kann, kann ein Methodenaufruf verwendet werden, der diesen Wert zurückliefert.**

**Statt**

```
int x = 3 + 70;
```

**sagen Sie**

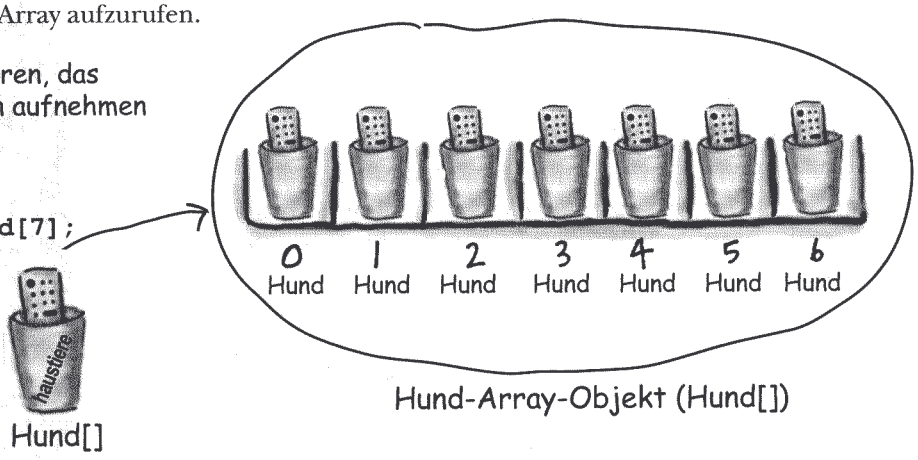
```
int x = 3 + eins.getGröße();
```

# Wie verhalten sich Objekte in einem Array?

Wie jedes andere Objekt. Der einzige Unterschied besteht in der Art, wie Sie an sie *herankommen*. Mit anderen Worten – wie Sie an die Fernsteuerung kommen. Versuchen wir mal, Methoden auf den Hund-Objekten in einem Hund-Array aufzurufen.

**1** Ein Hund-Array deklarieren, das sieben Hund-Referenzen aufnehmen soll.

```
Hund[] haustiere;
haustiere = new Hund[7];
```

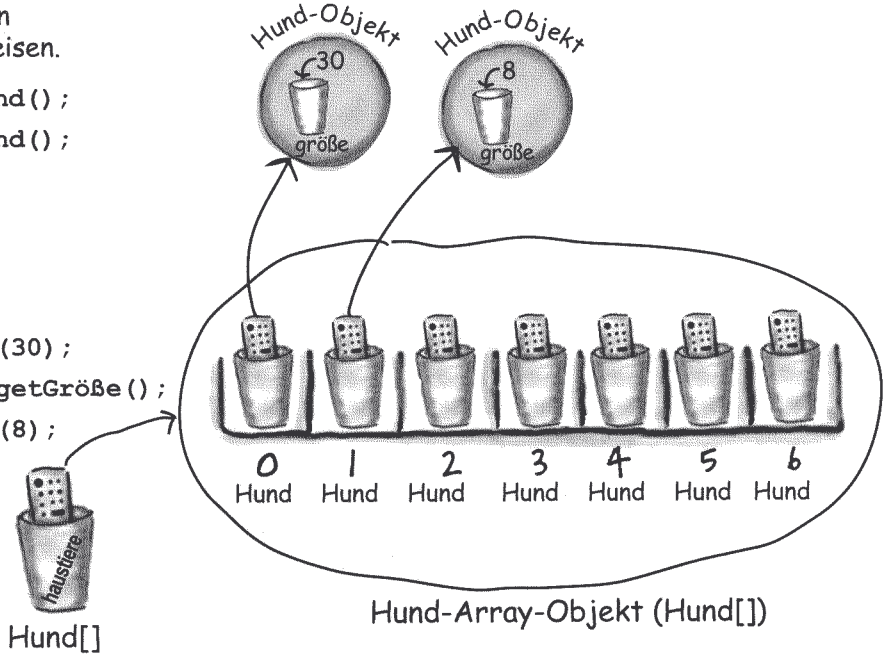



**2** Zwei neue Hund-Objekte erzeugen und diese den ersten beiden Elementen des Arrays zuweisen.

```
haustiere[0] = new Hund();
haustiere[1] = new Hund();
```

**3** Methoden auf den beiden Hund-Objekten aufrufen.

```
haustiere[0].setGröße(30);
int x = haustiere[0].getGröße();
haustiere[1].setGröße(8);
```



# Instanzvariablen deklarieren und initialisieren

Sie wissen bereits, dass bei einer Variablendeklaration zumindest ein Name und ein Typ erforderlich sind:

```
int größe;
String name;
```

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

```
int größe = 420;
String name = "Donny";
```

Aber was passiert, wenn Sie eine Getter-Methode aufrufen, wenn die entsprechende Instanzvariable nicht initialisiert wurde? Anders gesagt, was ist der *Wert* einer Instanzvariablen, *bevor* Sie sie initialisieren?

**Instanzvariablen erhalten immer einen Default-Wert. Auch wenn Sie einer Instanzvariablen nicht explizit einen Wert zuweisen oder keine Setter-Methode aufrufen, hat die Instanzvariable doch einen Wert!**

| | |
|---|---|
| Ganzzahlige Typen | 0 |
| Fließkommatypen | 0.0 |
| Boolesche Werte | false |
| Referenzen | null |

```
class ArmerHund {

   private int größe;
   private String name;

   public int getGröße() {
      return größe;
   }
   public String getName() {
      return name;
   }

}

public class ArmerHundTestlauf {
   public static void main (String[] args) {
      ArmerHund eins = new ArmerHund();
      System.out.println("Die Größe des Hundes ist " + eins.getGröße());
      System.out.println("Der Name des Hundes ist  " + eins.getName());
   }
}
```

Zwei Instanzvariablen deklarieren, ohne ihnen einen Wert zuzuweisen.

Was liefern die zurück??

Was meinen Sie? Lässt sich das überhaupt kompilieren?

```
Datei Bearbeiten Fenster Hilfe TierarztRufen
% java ArmerHundTestlauf
Die Größe des Hundes ist 0
Der Name des Hundes ist null
```

Sie müssen Instanzvariablen nicht initialisieren, weil sie immer einen Default-Wert haben. Elementare Zahlentypen (einschließlich char) haben den Wert 0, Boolesche Werte false und Objektreferenzvariablen null.

(Denken Sie daran, null meint einfach nur eine Fernsteuerung, die nichts kontrolliert/auf nichts programmiert ist. Eine Referenz, aber kein Objekt.)

# Der Unterschied zwischen Instanzvariablen und lokalen Variablen

**1** **Instanzvariablen** werden in einer Klasse deklariert, aber nicht in einer Methode.

```
class Pferd {
   private double höhe = 15.2;
   private String rasse;
   // mehr Code ...
}
```

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

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

   public int addieren() {
      int summe = a + b;
      return summe;
   }
}
```

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

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

Lässt sich nicht kompilieren!! Sie können x ohne einen Wert deklarieren, aber wenn Sie dann versuchen, es zu VERWENDEN, knallt dem Compiler eine Sicherung raus.

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

**Lokale Variablen erhalten KEINEN Default-Wert! Der Compiler beschwert sich, wenn Sie versuchen, eine lokale Variable zu verwenden, bevor die Variable initialisiert wurde.**

## Es gibt keine Dummen Fragen

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

A: Methodenparameter sind praktisch das Gleiche wie lokale Variablen – sie werden *in* der Methode deklariert (na ja, technisch gesehen werden sie in der *Argumentliste* der Methode deklariert und nicht im *Body* der Methode, aber es sind trotzdem lokale Variablen und keine Instanzvariablen). Aber es kann nicht sein, dass Methodenparameter nicht initialisiert sind. Deswegen erhalten Sie nie einen Compiler-Fehler, der Ihnen sagt, dass eine Parametervariable eventuell nicht initialisiert wurde.

Das liegt jedoch daran, dass der Compiler Ihnen eine Fehlermeldung präsentiert, wenn Sie versuchen, eine Methode aufzurufen, ohne ihr die Argumente zu senden, die sie benötigt. Parameter sind also IMMER initialisiert, weil der Compiler sichert, dass Methoden immer mit den Argumenten aufgerufen werden, die den Parametern entsprechen, die für die Methode deklariert sind, und die Argumente (automatisch) den Parametern zugewiesen werden.

# Variablen vergleichen (elementare Typen oder Referenztypen)

Es kann vorkommen, dass Sie wissen möchten, ob zwei *elementare Werte* gleich sind. Das ist nicht so schwer. Verwenden Sie einfach den ==-Operator. Manchmal möchten Sie auch wissen, ob zwei Referenzvariablen auf dasselbe Objekt auf dem Heap verweisen. Das ist ebenfalls einfach. Verwenden Sie wieder den ==-Operator. Aber manchmal wollen Sie wissen, ob zwei *Objekte* gleich sind. Und dafür brauchen Sie die Methode .equals(). Die Form der Gleichheit von Objekten ist vom Typ der Objekte abhängig. Ein Beispiel: Wenn zwei unterschiedliche Strings die gleichen Zeichen enthalten (»Suppenlöffel« beispielsweise), dann sind sie der Bedeutung nach gleichwertig, auch wenn es sich um zwei unterschiedliche Objekte auf dem Heap handelt. Aber was ist mit Hunden? Möchten Sie zwei Hunde als gleich betrachten, wenn sie die gleiche Größe und das gleiche Gewicht haben? Wahrscheinlich nicht. Ob zwei unterschiedliche Objekte als gleich behandelt werden, ist also davon abhängig, was für diesen bestimmten Typ Objekt sinnvoll ist. Das Konzept der Objektgleichheit werden wir uns in späteren Kapiteln (und Anhang B) noch genauer ansehen. Für den Augenblick müssen wir nur verstehen, dass der ==-Operator *nur* verwendet wird, um die Bits in zwei Variablen zu vergleichen. *Was* diese Bits repräsentieren, spielt keine Rolle. Die Bits sind entweder gleich oder nicht gleich.

**Verwenden Sie ==, um zwei elementare Werte zu vergleichen oder zu prüfen, ob zwei Referenzen auf dasselbe Objekt verweisen.**

**Verwenden Sie die Methode equals(), um zu prüfen, ob zwei unterschiedliche Objekte gleich sind.**

**(Wie zwei unterschiedliche String-Objekte, die beide die Zeichen in »Fred« repräsentieren.)**

### Verwenden Sie den ==-Operator, um elementare Werte zu vergleichen.

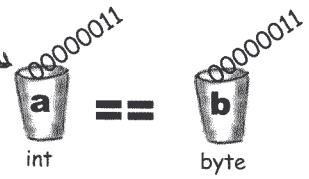
Der ==-Operator kann verwendet werden, um beliebige Variablen kompatibler Typen zu vergleichen, und vergleicht einfach nur die Bits.

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

```
int a = 3;
byte b = 3;
if (a == b) { // true }
```

(Es gibt mehr Nullen auf der linken Seite des int, aber das interessiert uns hier nicht.)

Die Bitmuster sind gleich, also sind die beiden Werte für == gleich.

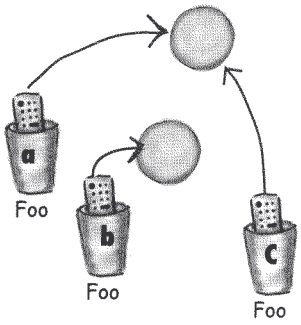



### Verwenden Sie den ==-Operator außerdem, um zu prüfen, ob zwei Referenzen gleich sind (das bedeutet, dass sie auf das gleiche Objekt auf dem Heap verweisen).

Denken Sie daran, dass sich der ==-Operator nur für das Bitmuster in der Variablen interessiert. Die Regeln sind die gleichen, egal ob eine Variable ein Referenztyp oder einen elementaren Typ repräsentiert. Der ==-Operator liefert also true zurück, wenn zwei Referenzvariablen auf dasselbe Objekt verweisen! In diesem Fall wissen wir nicht, was das Bitmuster ist (weil es von der JVM abhängig ist und vor uns verborgen wird), aber wir *wissen*, dass es, egal wie es aussieht, *für Referenzen auf dasselbe Objekt gleich ist.*

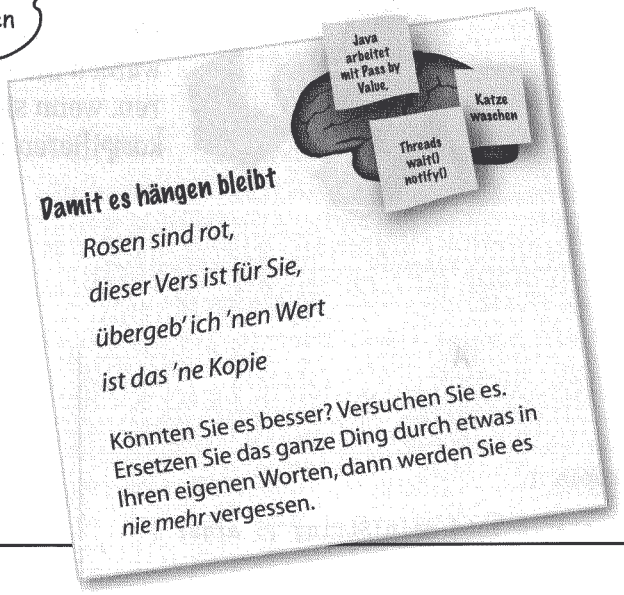
```
Foo a = new Foo();
Foo b = new Foo();
Foo c = a;
if (a == b) { // false }
if (a == c) { // true }
if (b == c) { // false }
```

Die Bitmuster für a und c sind gleich, also sind sie für == gleich.




a == c ist true

a == b ist false




## Spitzen Sie Ihren Bleistift

### Was ist zulässig?

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

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

```
int berechneFläche(int höhe, int breite) {
   return höhe * breite;
}
```

```
int a = berechneFläche(7, 12);
short c = 7;
berechneFläche(c,15);
int d = berechneFläche(57);
berechneFläche (2,3);
long t = 42;
int f = berechneFläche(t,17);
int g = berechneFläche();
berechneFläche();
byte h = berechneFläche(4,20);
int j = berechneFläche(2,3,5);
```

# Spielen Sie Compiler

Alle Java-Dateien auf dieser Seite stellen eine vollständige Quelldatei dar. Ihr Job ist es, Compiler zu spielen und festzustellen, ob sich diese Dateien kompilieren lassen. Wie würden Sie sie reparieren, wenn sie sich nicht kompilieren lassen?

## A

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

## B

```
class Uhr {
  String zeit;

  void setZeit(String z) {
    zeit = z;
  }

  void getZeit() {
    return zeit;
  }
}

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

    Uhr u = new Uhr();

    u.setZeit("1245");
    String heute = u.getZeit();
    System.out.println("Zeit: " + heute);
  }
}
```

# Wer bin ich?

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

**Heute Abend anwesend:**

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

**Eine Klasse kann eine beliebige Anzahl davon haben.** _____

**Eine Methode kann nur eins davon haben.** _____

**Können implizit befördert werden.** _____

**Ich ziehe es vor, dass meine Instanzvariablen privat sind.** _____

**Eigentlich bedeutet es »eine Kopie erstellen«.** _____

**Sie sollten nur von Settern aktualisiert werden.** _____

**Davon kann eine Methode mehrere haben.** _____

**Ich liefere etwas zurück.** _____

**Ich sollte bei Instanzvariablen nicht verwendet werden.** _____

**Ich kann viele Argumente haben.** _____

**Per Definition nehme ich nur ein Argument.** _____

**Ich helfe beim Aufbau einer Kapselung.** _____

**Ich fliege immer solo.** _____

## Vermischte Nachrichten

Unten ist ein kurzes Java-Programm aufgeführt. Ein Block des Programms fehlt. Ihre Herausforderung ist es, **die möglichen Codeblöcke** (unten links) **den Ausgaben zuzuordnen**, die Sie sähen, wenn der Block eingefügt würde.

Nicht alle Ausgabezeilen werden verwendet, und einige Ausgaben können mehrfach verwendet werden. Ziehen Sie Linien, die die Codeblock-Kandidaten mit den passenden Kommandozeilenausgaben verbinden. (Die Antworten finden Sie am Ende des Kapitels.)

```
public class MischMasch4 {
  int zähler = 0;
  public static void main(String [] args) {
    int anzahl = 0;
    MischMasch4[] m4a =new MischMasch4[20];
    int x = 0;
    while (          ) {
      m4a[x] = new MischMasch4();
      m4a[x].zähler = m4a[x].zähler + 1;
      anzahl = anzahl + 1;
      anzahl = anzahl
               + m4a[x].vielleichtNeu(x);
      x = x + 1;
    }
    System.out.println(anzahl + " "
                       + m4a[1].zähler);
  }

  public int vielleichtNeu(int index) {
    if (             ) {
      MischMasch4 m4 = new MischMasch4();
      m4.zähler = m4.zähler + 1;
      return 1;
    }
    return 0;
  }
}
```

**Kandidaten:**

```
x < 9
index < 5
```

```
x < 20
index < 5
```

```
x < 7
index < 7
```

```
x < 19
index < 1
```

**Mögliche Ausgaben:**

```
14 7
```

```
9 5
```

```
19 1
```

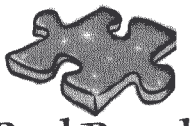
```
14 1
```

```
25 1
```

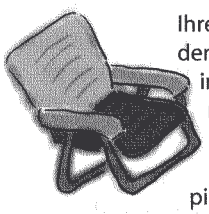
```
7 7
```

```
20 1
```

```
20 5
```



## Pool-Puzzle



Ihre ***Aufgabe*** ist es, die Codeschnipsel aus dem Pool zu nehmen und sie in die Lücken im Code einzusetzen. Jedes Schnipsel darf **nur einmal** verwendet werden, und Sie werden nicht alle benötigen. Ihr ***Ziel*** ist es, eine Klasse zu erstellen, die sich kompilieren und ausführen lässt und die hier gezeigte Ausgabe erzeugt.

**Ausgabe**

```
Datei Bearbeiten Fenster Hilfe BauchLandung
%java Puzzle4
Ergebnis 543345
```

```
public class Puzzle4 {
  public static void main(String [] args)
  {
    _____________________________________
    int y = 1;
    int x = 0;
    int erg = 0;
    while (x < 6) {
      _____________________________
      _____________________________
      y = y * 10;
      ___________________
    }
    x = 6;
    while (x > 0) {
      ___________________
      erg = erg + _____________________
    }
    System.out.println("Ergebnis " + erg);
  }
}
class ___________ {
  int ivar;
  _________ _______ machZeug(int __________)
  {
    if (ivar > 100) {
      return _____________________________
    } else {
      return _____________________________
    }
  }
}
```

**Hinweis: Jedes Schnipsel aus dem Pool kann nur einmal verwendet werden!**

machZeug(x);
obs.machZeug(x);
obs[x].machZeug(faktor);
ivar = x;
obs.ivar = x;
obs[x].machZeug(x);
obs[x].ivar = x;
obs[x].ivar = y;
ivar + faktor;
ivar * (2 + faktor);
ivar * (5 - faktor);
ivar * faktor;
Puzzle4
Puzzle4b
Puzzle4b( )
int
short
ivar
faktor
public
private
Puzzle4 [ ] obs = new Puzzle4[6];
Puzzle4b [ ] obs = new Puzzle4b[6];
Puzzle4b [ ] obs = new Puzzle4[6];
x = x + 1;
x = x - 1;
obs [x] = new Puzzle4b(x);
obs [ ] = new Puzzle4b( );
obs [x] = new Puzzle4b( );
obs = new Puzzle4b( );

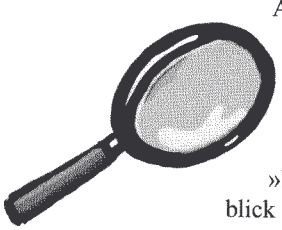
## Unruhige Zeiten in Stim-City

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

Kurzkrimi

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

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



»Ich fürchte, so einfach ist das nicht, Jai. Buchanan hier sagt mir, es hieße, dass du im Augenblick auf J37NE bist«, insistierte Leveler. »Neural-Edition? Klar, ich spiel etwas herum. Und?«, antworte Jai mit einem leicht mulmigen Gefühl in der Magengegend. »Über Neural-Edition lass ich die Stim-Junkies wissen, wo es die nächste Lieferung gibt«, erklärte Leveler. »Das Problem ist, dass ein paar Stim-Junkies so lang drauf blieben, bis sie herausgefunden hatten, wie man sich in meine Waren-Haus-Datenbank hackt.« »Ich brauch jemanden, der schnell denken kann, jemanden wie dich, Jai, der einen Blick auf meine StimDrop J37NE-Klasse wirft – Methoden, Instanzvariablen und den ganzen Salat – und herausfindet, wie sie hereinkommen. Sollte ...« »HE!«, rief Buchanan, »ich will nicht, dass Hacker-Abschaum wie der in meinem Code herumschnüffelt!« »Ganz ruhig, großer Junge!« Jai sah seine Chance. »Ich bin sicher, du hast mit deinen Zugriffsmodi ganze Arbeit geleistet ...« »Willst du mir etwa was erklären – Bit-Fummler!«, rief Buchanan, »ich hab alle Methoden auf Junkie-Level öffentlich gemacht, damit sie auf die Drop-Site-Daten zugreifen können, aber alle kritischen WarenHaus-Methoden privat gelassen. Keiner von draußen kann auf diese Methoden zugreifen, Freund!«

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

### Welchen Verdacht hat Jai?

### Wird er Levelers Wagen mit heilen Knochen verlassen?

## Lösungen zu den Übungen

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

**B**

```
class Uhr {
  String zeit;
  void setZeit(String z) {
    zeit = z;
  }
  String getZeit() {
    return zeit;
  }
}

class UhrTestlauf {
  public static void main(String [] args) {
    Uhr c = new Uhr();
    c.setZeit("1245");
    String heute = c.getZeit();
    System.out.println("Zeit: " + heute);
  }
}
```

Hinweis: Getter-Methoden haben definitionsgemäß einen Rückgabewert.

---

| | |
|---|---|
| Eine Klasse kann eine beliebige Anzahl davon haben. | Instanzvariablen, Getter, Setter, Methoden |
| Eine Methode kann nur eins davon haben. | return |
| Können implizit befördert werden. | return, Argument |
| Ich ziehe es vor, dass meine Instanzvariablen privat sind. | Kapselung |
| Eigentlich bedeutet es »eine Kopie erstellen«. | Pass by Value |
| Sie sollten nur von Settern aktualisiert werden. | Instanzvariablen |
| Davon kann eine Methode mehrere haben. | Argument |
| Ich liefere etwas zurück. | Getter, return |
| Ich sollte bei Instanzvariablen nicht verwendet werden. | public |
| Ich kann viele Argumente haben. | Methoden |
| Per Definition nehme ich nur ein Argument. | Setter |
| Ich helfe beim Aufbau einer Kapselung. | Getter, Setter, public, private |
| Ich fliege immer solo. | return |

# Puzzle-Lösungen

```
public class Puzzle4 {
  public static void main(String [] args) {
    Puzzle4b [ ] obs = new Puzzle4b[6];
    int y = 1;
    int x = 0;
    int erg = 0;
    while (x < 6) {
      obs[x] = new Puzzle4b( );
      obs[x] . ivar = y;
      y = y * 10;
      x = x + 1;
    }
    x = 6;
    while (x > 0) {
      x = x - 1;
      erg = erg + obs[x].machZeug(x);
    }
    System.out.println("Ergebnis " + erg);
  }
}
class Puzzle4b {
  int ivar;
  public int machZeug(int faktor) {
    if (ivar > 100) {
      return ivar * faktor;
    } else {
      return ivar * (5 - faktor);
    }
  }
}
```

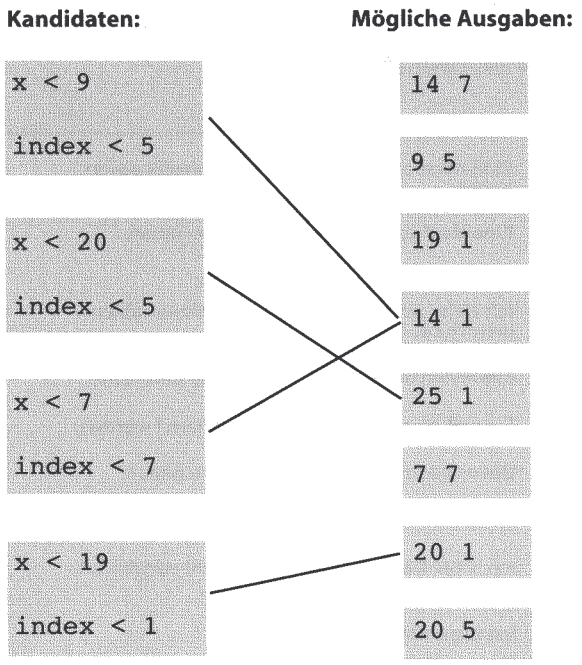
**Ausgabe**

```
Datei Bearbeiten Fenster Hilfe BauchLandung
%java Puzzle4
Ergebnis 543345
```

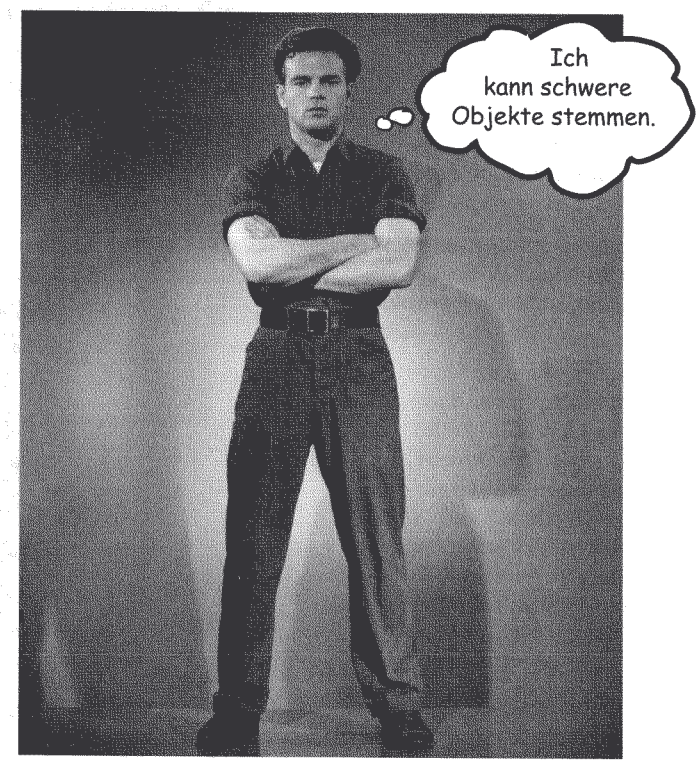
Antwort auf den Kurzkrimi ...

Jai wusste, dass Buchanan nicht unbedingt das schärfste Messer im Block war. Als Buchanan über seinen Code sprach, sagte er kein Wort über seine Instanzvariablen. Jai hatte den Verdacht, dass Buchanan zwar seine Methoden richtig behandelt, aber versäumt hatte, seine Instanzvariablen `privat` zu machen. Dieses Versäumnis konnte Leveler schnell Tausende gekostet haben.

---



# Methoden mit Superkraft




**Jetzt werden wir unseren Methoden etwas mehr Muskelkraft geben.** Wir haben uns an Variablen versucht, haben mit ein paar Objekten gespielt und haben etwas Code geschrieben. Aber wir waren noch zu schwach. Wir brauchen mehr Werkzeuge. Wie **Operatoren**. Wir brauchen mehr Operatoren, damit wir Dinge tun können, die etwas interessanter sind als *bellen*. Und **Schleifen**. Wir brauchen Schleifen. Mehr als diese schwächlichen *while*-Schleifen! Wir brauchen ***for***-Schleifen, wenn wir richtig ernsthaft werden wollen. Es könnte auch nützlich sein, **Zufallszahlen zu generieren**. Oder **einen String in ein int umzuwandeln**, ja, das wäre cool. Sollten wir besser auch lernen. Und warum lernen wir nicht all den Kram, während wir was Echtes *schreiben*, um zu erfahren, wie es ist, wenn man ein Programm von der Pike auf schreiben (und testen) muss. **Vielleicht ein Spiel**, etwas wie Schiffe versenken. Das ist eine richtige Gewichtheber-Aufgabe. Wir brauchen *zwei* Kapitel, um sie zu vollenden. In diesem Kapitel werden wir eine einfache Version erstellen. Eine mächtigere De-luxe-Version bauen wir dann in Kapitel 6.

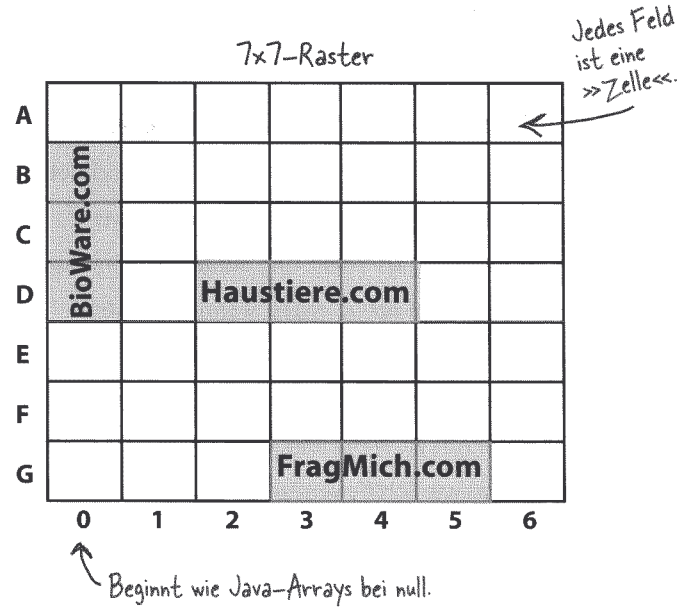
# Bauen wir ein Spiel im Schiffe-versenken-Stil: »Dot-Com versenken«

Sie müssen sich mit dem Computer schlagen. Aber anders als beim richtigen Schiffeversenken platzieren Sie in diesem Spiel keine eigenen Schiffe. Stattdessen ist es Ihre Aufgabe, die Schiffe des Computers mit möglichst wenigen Versuchen zu versenken. Außerdem versenken wir keine Schiffe. Wir versenken Dot-Coms. (Sehen Sie? Das Buch hat doch einen praktischen Nutzen. Sie werden die Kosten des Buchs schon wieder einspielen!).

**Ziel**: Alle Dot-Coms des Computers mit möglichst wenigen Versuchen zu versenken. Am Ende erhalten Sie eine Bewertung oder eine Einstufung, die auf Ihrer Leistung basiert.

**Setup**: Wenn das Spielprogramm gestartet wird, platziert der Computer drei Dot-Coms auf einem **virtuellen 7x7-Raster**. Ist das abgeschlossen, fordert das Spiel Sie zu Ihrem ersten Rateversuch auf.

**Wie Sie spielen:** Wir wissen noch nicht, wie man ein GUI aufbaut, also arbeitet diese Version auf der Kommandozeile. Der Computer fordert Sie auf, einen Tipp (eine Zelle) abzugeben, den Sie auf der Kommandozeile (als »A3«, »C5« usw.) eingeben. Als Antwort auf Ihren Tipp sehen Sie auf der Kommandozeile als Ergebnis entweder »Treffer«, »Vorbei« oder »Sie haben Haustiere.com versenkt« (oder was immer das Dot-Com-Unternehmen der Stunde ist). Wenn Sie alle drei Dot-Coms in den großen 404-Himmel geblasen haben, endet das Spiel damit, dass Ihre Bewertung ausgegeben wird.




**Sie werden das Spiel Dot-Com versenken mit einem 7x7-Raster und drei Dot-Coms aufbauen. Jedes Dot-Com nimmt drei Zellen ein.**

## Teil eines Spielablaufs

```
Datei Bearbeiten Fenster Hilfe Verkaufen
%java DotComVersenken
Geben Sie einen Rateversuch ein  D4
Treffer
Geben Sie einen Rateversuch ein  D5
Vorbei
Geben Sie einen Rateversuch ein  E4
Vorbei
Geben Sie einen Rateversuch ein  D3
Treffer
Geben Sie einen Rateversuch ein  D2
Grrr! Sie haben Haustiere.com versenkt : (
Versenkt
Geben Sie einen Rateversuch ein  B2
Vorbei
Geben Sie einen Rateversuch ein  G1
Vorbei
Geben Sie einen Rateversuch ein  G4
Treffer
Geben Sie einen Rateversuch ein  F4
Vorbei
Geben Sie einen Rateversuch ein  G3
Treffer
Geben Sie einen Rateversuch ein  G5
Grrr! Sie haben FragMich.com versenkt : (
Versenkt
Geben Sie einen Rateversuch ein  B6
Vorbei
Geben Sie einen Rateversuch ein  G0
```

# Zuerst ein abstrakter Entwurf

Wir wissen, dass wir Klassen und Methoden brauchen, aber welche sollen das sein? Um diese Frage zu beantworten, benötigen wir weitere Informationen dazu, was das Spiel tun soll. Dazu müssen wir erst den allgemeinen Ablauf des Spiels herausarbeiten. Hier ist der Grundgedanke:

**1** Der Benutzer startet das Spiel

- **A** Das Spiel erzeugt drei Dot-Coms.
- **B** Das Spiel platziert die drei Dot-Coms auf einem virtuellen Raster.

**2** Der Spielablauf beginnt

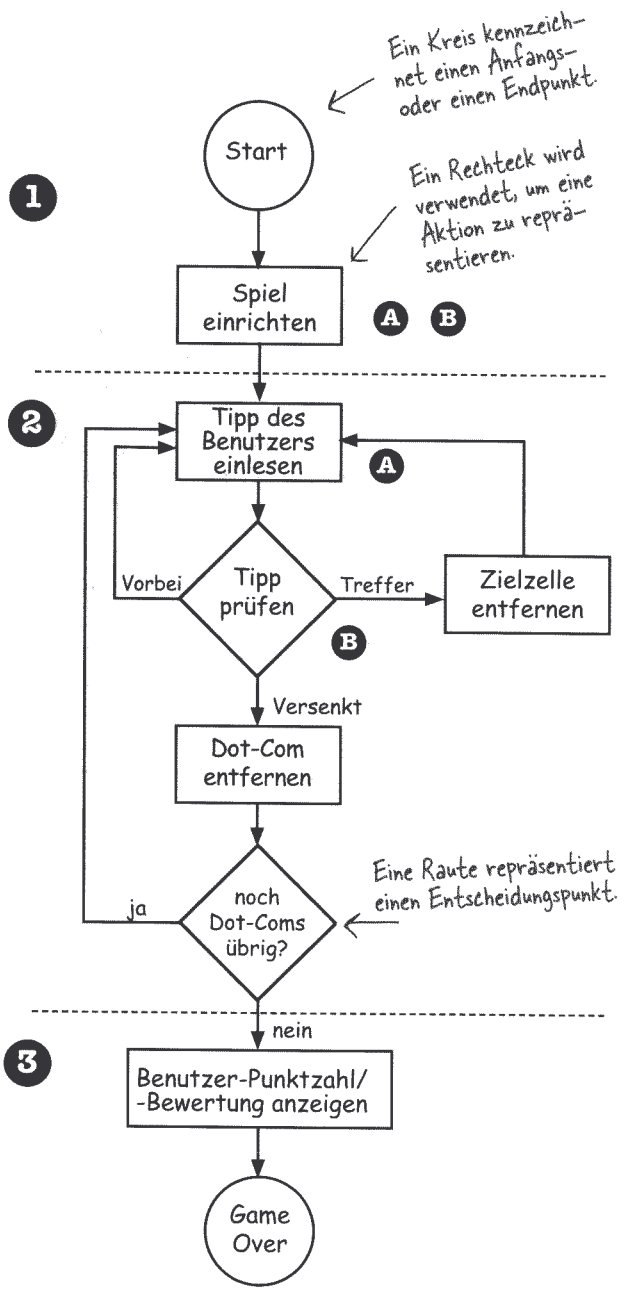
Die folgenden Schritte wiederholen, bis es keine Dot-Coms mehr gibt:

- **A** Den Benutzer zur Eingabe eines Rateversuchs auffordern (»A2«, »C0« usw.)
- **B** Den Rateversuch des Spielers mit allen Dot-Coms vergleichen, um zu prüfen, ob das Ergebnis Treffer, Vorbei oder Versenkt ist. Die entsprechende Handlung vornehmen: bei Treffer die Zelle löschen, bei Versenkt das Dot-Com löschen.

**3** Das Spiel wird beendet

Dem Benutzer auf Grundlage der benötigten Versuche eine Bewertung geben.

Jetzt haben wir eine Vorstellung davon, welche Dinge das Programm tun muss. Der nächste Schritt ist herauszufinden, welche Art von **Objekten** wir benötigen, um die Arbeit zu verrichten. Denken Sie dabei eher wie Stefan als wie Harry. Konzentrieren Sie sich auf die ***Dinge*** im Programm statt auf die ***Prozeduren***.

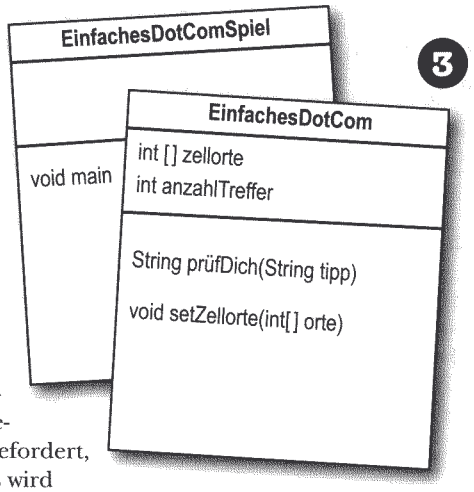



Wow. Ein echtes Flussdiagramm.

# »EinfachesDotComSpiel« – eine sanftere Einführung

Es sieht so aus, als benötigten wir mindestens zwei Klassen: eine Spiel-Klasse und eine DotCom-Klasse. Aber bevor wir das vollständige Spiel ***Dot-Com versenken*** aufbauen, beginnen wir mit einer reduzierten, vereinfachten Version, ***EinfachesDotComSpiel.*** Die einfachere Version werden wir in *diesem* Kapitel aufbauen, im *nächsten* Kapitel werden wir dann die De-luxe-Version nachschieben.

Alles in diesem Spiel ist einfacher. Statt in einem 2-D-Raster verbergen wir das Dot-Com nur in einer einzigen *Reihe.* Und wir verwenden auch nicht *drei* Dot-Coms, sondern nur *eins.* Das Ziel bleibt allerdings das gleiche. Das Spiel muss also immer noch eine DotCom-Instanz erstellen, ihr einen Ort irgendwo in der Zeile zuweisen, Benutzereingaben abfragen und enden, wenn alle Zellen des Dot-Coms getroffen wurden. Die vereinfachte Version des Spiels verschafft uns einen großen Vorteil für den Aufbau des vollständigen Spiels. Wenn wir dieses kleine Spiel zum Laufen bringen, können wir es für das folgende komplexere Spiel erweitern.

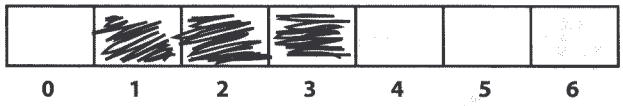



In dieser einfachen Version hat die Spielklasse keine Instanzvariablen. Der gesamte Spielcode befindet sich in der main()-Methode. Anders gesagt, wenn das Programm gestartet wird und main() zu laufen beginnt, wird die eine DotCom-Instanz erstellt, für sie ein Ort (drei aufeinander folgende Zellen in der einen virtuellen, aus sieben Zellen bestehenden Reihe) ausgewählt, der Benutzer zu einem Rateversuch aufgefordert, der Rateversuch geprüft, und alles wird so lange wiederholt, bis die drei Zellen getroffen wurden.

Denken Sie daran, dass die virtuelle Reihe ... *virtuell* ist. Oder anders ausgedrückt, es gibt sie an keiner Stelle des Programms. So lange Spiel und Benutzer beide wissen, dass sich das Dot-Com in drei aufeinander folgenden Zellen von sieben möglichen (bei null beginnenden) Zellen befindet, muss die Reihe selbst im Code nicht repräsentiert werden. Vielleicht sind Sie versucht, ein Array mit sieben ints zu erstellen und das Dot-Com dann drei der sieben Elemente des Arrays zuzuweisen. Aber das müssen Sie nicht tun. Wir brauchen bloß ein Array, das die drei Zellen festhält, die das Dot-Com besetzt.

1. **Das Spiel beginnt,** erzeugt EIN Dot-Com und gibt ihm einen Ort auf drei Zellen in einer einzigen Zeile mit sieben Zellen.

   Statt »A2«, »C4« und so weiter sind die Orte einfach nur Ganzzahlen. In diesem Bild sind beispielsweise 1, 2 und 3 die Zellorte:




2. **Der Spielablauf beginnt.** Den Benutzer zur Eingabe eines Rateversuchs auffordern, dann prüfen, ob er eine der drei Zellen des Dot-Com getroffen hat. Wenn es einen Treffer gab, die Variable anzahlTreffer inkrementieren.

3. **Das Spiel endet,** wenn alle drei Zellen getroffen wurden (der Wert der Variablen anzahlTreffer ist 3), und sagt dem Benutzer, wie viele Versuche er brauchte, um das Dot-Com zu versenken.

### Die ganze Programminteraktion

```
Datei Bearbeiten Fenster Hilfe Zerstören
% java EinfachesDotComSpiel
Geben Sie eine Zahl ein 2
Treffer
Geben Sie eine Zahl ein 3
Treffer
Geben Sie eine Zahl ein 4
Vorbei
Geben Sie eine Zahl ein 1
Versenkt
Sie haben 4 Versuche benötigt
```

# Eine Klasse entwickeln

Als Programmierer haben Sie wahrscheinlich eine Methode/einen Prozess/eine Herangehensweise beim Schreiben von Code. Haben wir auch. Die Schritte, die wir durchführen, sind so entworfen, dass Sie erkennen (und lernen), was wir uns denken, wenn wir uns durch die Programmierung einer Klasse arbeiten. Das Verfahren entspricht nicht notwendigerweise dem Verfahren, das wir (oder *Sie*) verwenden, wenn wir im wahren Leben Code schreiben. Im wahren Leben folgen Sie natürlich dem Verfahren, das Ihnen Ihre persönlichen Vorlieben, das Projekt oder ein Arbeitgeber vorgeben. Aber wir können so ziemlich alles machen, was wir wollen. Und wenn wir eine Java-Klasse als »Lernmuster« schreiben, machen wir üblicherweise Folgendes:

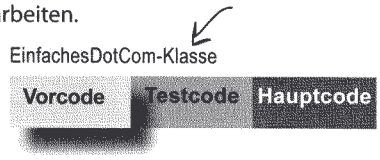
- ☐ Wir versuchen herauszufinden, was die Klasse *tun* soll.
- ☐ Wir machen eine Liste der **Instanzvariablen und Methoden**.
- ☐ Wir schreiben **Vorcode** für die Methoden. (Das werden Sie gleich sehen.)
- ☐ Wir schreiben **Testcode** für die Methoden.
- ☐ Wir **implementieren** die Klasse.
- ☐ Wir **testen** die Methoden.
- ☐ Wir **debuggen** und **reimplementieren** nach Bedarf.
- ☐ Wir drücken unsere Dankbarkeit dafür aus, dass wir unsere so genannte *Lernmuster*-Anwendung nicht an unseren echten lebenden Benutzern ausprobieren müssen.

**Zeit, ein paar Neuronen zu massieren.**

**Wie würden Sie entscheiden, welche Klasse oder Klassen *zuerst* erstellt werden müssen, wenn Sie ein Programm schreiben? Vorausgesetzt, dass selbst die winzigsten Programme mehr als eine Klasse benötigen: Wo fangen Sie an (wenn Sie guten OO-Prinzipien folgen und *keine* Klassen haben, die viele unterschiedliche Aufgaben übernehmen)?**

## Die drei Dinge, die wir für jede Klasse schreiben:

Diese Leiste wird auf den nächsten Seiten angezeigt, um Ihnen zu sagen, an welchem Teil Sie gerade arbeiten. Wenn Sie dieses Bild am Anfang einer Seite sehen, heißt das, dass Sie am Vorcode für die Klasse EinfachesDotCom arbeiten.




### Vorcode

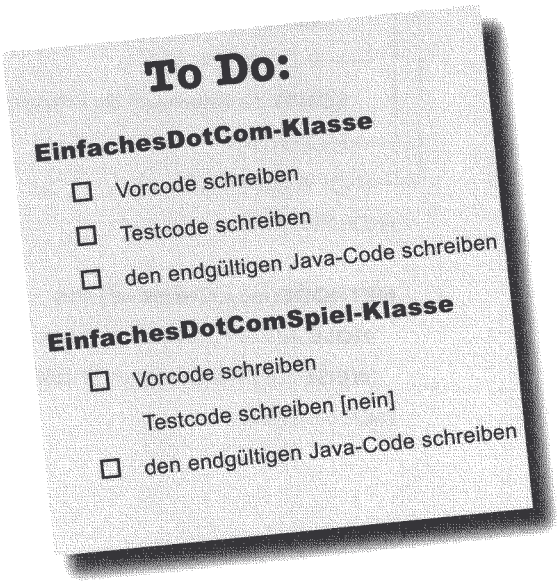
Eine Form von Pseudocode, die Ihnen helfen soll, sich auf die Logik zu konzentrieren, ohne sich mit der Syntax herumschlagen zu müssen.

### Testcode

Eine Klasse oder Methoden, die den Hauptcode testen und prüfen, ob er macht, was er machen soll.

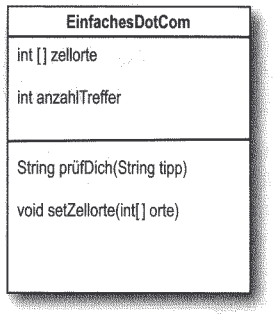
### Hauptcode

Die eigentliche Implementierung der Klasse. Hier schreiben Sie den echten Java-Code.



## EinfachesDotCom-Klasse

Vorcode | Testcode | Hauptcode




Sie bekommen eine Vorstellung davon, wie Vorcode (unsere Version von Pseudocode) funktioniert, wenn Sie dieses Beispiel durchlesen. Es ist so eine Art Zwischenstufe zwischen echtem Java-Code und einer einfachen deutschen Beschreibung der Klasse. Die meisten Vorcodes enthalten drei Teile: Instanzvariablen-Deklarationen, Methoden-Deklarationen, Methodenlogik. Der wichtigste Teil des Vorcodes ist die Methodenlogik, weil er definiert, *was* passieren muss, und von uns später in das *Wie* übersetzt wird, wenn wir tatsächlich den Methodencode schreiben.

**DEKLARATION:** Ein ***int-Array***, das die Zellorte festhält. Nenn es ***zellorte***.

**DEKLARATION:** Ein ***int***, das die Anzahl von Treffern festhält. Nenn es ***anzahlTreffer*** und **SETZE** es auf 0.

---

**DEKLARATION:** Eine ***prüfDich()***-Methode, die einen ***String*** für den Tipp des Benutzers (»1«, »3« usw.) entgegennimmt, diesen prüft und ein Ergebnis zurückliefert, das »Treffer«, »Vorbei« oder »Versenkt« repräsentiert.

**DEKLARATION:** Eine ***setZellorte()***-Setter-Methode, die ein ***int-Array*** nimmt (das drei Zellorte als ***ints*** (z. B. 2, 3, 4) enthält).

---

**METHODE:** *String prüfDich(String tipp)*

**HOLE** den Rateversuch des Benutzers als String-Parameter.

**KONVERTIERE** den Rateversuch des Benutzers in ein ***int***.

**WIEDERHOLE** das mit jedem der Zellorte im int-Array.

***// VERGLEICHE*** den Rateversuch des Benutzers mit dem Zellort.

**WENN** der Rateversuch des Benutzers passt:

**INKREMENTIERE** die Anzahl von Treffern,

***// ERMITTLE***, ob es der letzte Zellort war:

**WENN** Anzahl der Treffer gleich 3, **LIEFER** »Versenkt« als Ergebnis

**SONST** nicht versenkt, also **LIEFER** »Treffer«

ENDE WENN

**SONST** der Rateversuch des Benutzers passt nicht, also **LIEFER** »Vorbei«

ENDE WENN

ENDE WIEDERHOLE

ENDE METHODE

**METHODE:** *void setZellorte( int[] orte)*

**HOLE** die Zellorte als ***int-Array***-Parameter

**WEISE** den Zellortparameter der zellorte-Instanzvariable **ZU**

ENDE METHODE

Vorcode | Testcode | Hauptcode

# Die Methodenimplementierungen schreiben

## Schreiben wir jetzt also den echten Methodencode und bringen wir das Baby ans Laufen.

Bevor wir mit dem Programmieren der Methoden beginnen, lassen Sie uns erst einmal einen Schritt zurückgehen und etwas Code schreiben, um die Methoden zu *testen*. Sie haben richtig gehört: Wir schreiben den Testcode, *bevor* es irgendetwas zu testen gibt!

Das Konzept, zuerst Testcode zu schreiben, ist eine der Praktiken des Extreme Programming (XP). Dies kann es für Sie leichter (und schneller) machen, Ihren Code zu schreiben, was aber nicht bedeutet, dass Sie unbedingt XP verwenden müssen. Aber uns gefällt daran, den Testcode zuerst zu schreiben. Und XP *klingt* einfach cool.

### Extreme Programming (XP)

Extreme Programming (XP) ist ein Neuankömmling in der Welt der Software-Entwicklungsmethodik. XP, das für viele »die Art ist, wie Programmierer wirklich arbeiten wollen«, kam in den späten 90er-Jahren auf und wurde von vielen Unternehmen, seien es Zwei-Personen-Klitschen oder auch die Ford Motor Company, übernommen. Die Zielrichtung von XP ist, dass der Kunde bekommt, was er haben will, auch wenn sich die Spezifikation spät im Entwicklungsprozess noch ändert.

XP basiert auf einem Satz bewährter Praktiken, die auf ihr Zusammenwirken hin entworfen sind. Allerdings suchen, wählen und übernehmen viele nur einen Teil der Regeln von XP. Zu diesen Praktiken gehören Dinge wie:

Machen Sie kleine, aber häufige Releases.

Entwickeln Sie in Iterationszyklen.

Bauen Sie nichts ein, was nicht in der Spezifikation steht (egal wie versucht Sie sind, Funktionalitäten »für die Zukunft« einzubauen).

Schreiben Sie den Testcode *zuerst*.

Keine Killer-Zeitpläne. Arbeiten Sie mit geregeltem Stundenplan.

Führen Sie Refactoring (Codeverbesserungen) durch, wann immer Sie eine Gelegenheit zur Verbesserung sehen.

Geben Sie nichts frei, das nicht alle Tests bestanden hat.

Setzen Sie realistische Zeitpläne, die auf kleinen Releases basieren.

Halten Sie es einfach.

Programmieren Sie paarweise und lassen Sie die Mitarbeiter rotieren, damit jeder so ziemlich alles über den Code weiß.

# Den Testcode für die Klasse EinfachesDotCom schreiben

Wir müssen Testcode schreiben, der ein EinfachesDotCom-Objekt erzeugen und seine Methoden ausführen kann. Bei der Klasse EinfachesDotCom interessieren wir uns eigentlich nur für die Methode *prüfDich()*, wir *müssen* allerdings auch die Methode *setZellorte()* implementieren, damit *prüfDich()* richtig laufen kann.

Werfen Sie einen genauen Blick auf den unten stehenden Testcode für die Methode *prüfDich()* (die Methode *setZellorte()* ist eine triviale Setter-Methode, um die wir uns deswegen nicht kümmern müssen, obwohl wir in einer »echten« Anwendung vielleicht gern eine robustere Setter-Methode hätten, die wir dann auch testen *müssten*).

Fragen Sie sich dann: »Wenn die Methode prüfDich() implementiert wäre, welchen Testcode könnte ich schreiben, der mir belegen würde, dass die Methode richtig funktioniert?«

## Auf Grundlage dieses Vorcodes:

**METHODE:** *String prüfDich(String tipp)*

**HOLE** den Rateversuch des Benutzers als String-Parameter.

**KONVERTIERE** den Rateversuch des Benutzers in ein ***int***.

**WIEDERHOLE** das mit jedem der Zellorte im ***int***-Array.

*// **VERGLEICHE*** den Rateversuch des Benutzers mit dem // Zellort.

**WENN** der Rateversuch des Benutzers passt:

**INKREMENTIERE** die Anzahl von Treffern,

*// **ERMITTLE***, ob es der letzte Zellort war:

**WENN** Anzahl der Treffer gleich 3, **LIEFER** »Versenkt« als Ergebnis

**SONST** nicht versenkt, also LIEFER »Treffer«

ENDE WENN

**SONST** der Rateversuch des Benutzers passt nicht, also **LIEFER** »Vorbei«

ENDE WENN

ENDE WIEDERHOLE

ENDE METHODE

## sollten folgende Dinge getestet werden:

1. Ein EinfachesDotCom-Objekt instantiieren.
2. Ihm einen Ort zuweisen (ein Array mit 3 ints wie {2,3,4}).
3. Einen String erzeugen, der einen Rateversuch des Benutzers repräsentiert (»2«, »0« usw.).
4. Die Methode prüfDich() aufrufen und den vermeintlichen Rateversuch übergeben.
5. Das Ergebnis ausgeben, um zu prüfen, ob es richtig ist (»bestanden« oder »fehlgeschlagen«).

## Es gibt keine Dummen Fragen

**F: Vielleicht ist mir da ja etwas entgangen, aber wie genau führe ich einen Test auf etwas aus, das es noch gar nicht gibt!?**

A: Gar nicht. Wir haben nie gefordert, dass Sie damit beginnen sollen, den Test *auszuführen*. Sie beginnen damit, dass Sie den Test *schreiben*. Während des Schreibens gibt es noch nichts, gegen das Sie ihn ausführen können. Wahrscheinlich können Sie ihn erst kompilieren, wenn Sie so genannten »Stub-Code« schreiben, der das Kompilieren ermöglicht, bei dem alle Tests aber noch fehlschlagen (beispielsweise null zurückliefern).

**F: Dann kapier ich immer noch nicht, was das soll. Warum warten wir nicht, bis der Code geschrieben ist, und zimmern dann den Testcode zusammen?**

A: Das Durchdenken (und Schreiben) des Testcodes hilft Ihnen dabei, ein klareres Bild davon zu bekommen, was die Methode selbst tun muss.

Sobald Ihr Implementierungscode fertig ist, haben Sie den Testcode zur Hand, der nur darauf wartet, ersteren zu prüfen. Außerdem *wissen* Sie genau, dass Sie es *nie* tun werden, wenn Sie es jetzt nicht tun. Es gibt immer irgendetwas zu tun, das interessanter ist.

Im Idealfall schreiben Sie etwas Testcode und dann *nur* den Implementierungscode, den Sie brauchen, um diesen Test zu bestehen. Dann schreiben Sie etwas *mehr* Testcode und *nur* den neuen Implementierungscode, den Sie brauchen, um *diesen* neuen Test zu bestehen. Bei jedem Testdurchlauf lassen Sie *alle* bereits geschriebenen Tests laufen, damit Sie immer nachweisen können, dass die neuesten Code-Erweiterungen den bereits getesteten Code nicht zerbrechen.

# Testcode für die Klasse EinfachesDotCom

```
public class EinfachesDotComTestlauf {

  public static void main (String[] args) {

    EinfachesDotCom dotCom = new EinfachesDotCom();

    int[] orte = {2,3,4};

    dotCom.setZellorte(orte);

    String tipp = "2";

    String ergebnis = dotCom.prüfDich(tipp);

    String testErgebnis = "Fehlgeschlagen";

    if (ergebnis.equals("Treffer") ) {

          testErgebnis = "Bestanden";

    }

    System.out.println(testErgebnis);

  }

}
```

Ein Einfaches-DotCom-Objekt instantiieren.

Ein int-Array für den Ort des Dot-Com erstellen (drei aufeinander folgende ints aus möglichen sieben).

Die Setter-Methode auf dem Dot-Com aufrufen.

Einen möglichen Benutzervorschlag abgeben.

Die Methode prüfDich() auf dem DotCom-Objekt aufrufen und ihr den möglichen Benutzervorschlag übergeben.

Wenn der simulierte Benutzervorschlag (2) »Treffer« zurückliefert, funktioniert die Methode.

Das Testergebnis ausgeben (Bestanden oder Fehlgeschlagen).

### Spitzen Sie Ihren Bleistift

Auf den nächsten paar Seiten werden wir die Klasse EinfachesDotCom implementieren und dann später wieder zu unserer Testklasse zurückkehren. Sehen Sie sich unseren Testcode an und überlegen Sie, was sonst noch hinzugefügt werden sollte. Testen wir in diesem Code irgendetwas *nicht*, was wir testen *sollten*? Notieren Sie unten Ihre Ideen (oder die entsprechenden Codezeilen):

# Die Methode prüfDich()

Es gibt keine perfekte Abbildung von Vorcode auf Java-Code. Sie werden ein paar Anpassungen sehen. Der Vorcode hat uns jedoch ein viel besseres Bild davon verschafft, *was* der Code tun muss, und jetzt müssen wir den Java-Code finden, der das *Wie* erledigen kann.

Behalten Sie im Hinterkopf, welche Teile dieses Codes Sie gern verbessern möchten (oder verbessern müssten). Zahlen wie (1) markieren Dinge (Syntax- und Sprach-Features), die Ihnen bisher noch nicht begegnet sind. Sie werden auf der gegenüberliegenden Seite erklärt.

**HOLE** den Rateversuch des Benutzers.
**KONVERTIERE** den Rateversuch des Benutzers in ein int.

**WIEDERHOLE** mit jeder Zelle im int-Array.
**WENN** der Rateversuch des Benutzers passt:

**INKREMENTIERE** die Anzahl von Treffern.

*// ERMITTLE*, ob es der letzte Zellort war:

**WENN** Anzahl der Treffer gleich 3

**LIEFER** »Versenkt« als Ergebnis

**SONST** nicht versenkt, also

**LIEFER** »Treffer«

**SONST**

**LIEFER** »Vorbei«

```
public String prüfDich(String stringTipp) {

  int tipp = Integer.parseInt(stringTipp);   (1)

  String ergebnis = "Vorbei";

  for (int zelle : zellorte) {   (2)

    if (tipp == zelle) {

      ergebnis = "Treffer";

      anzahlTreffer++;   (3)

      break;   (4)

    } // if-Ende

  } // for-Ende

  if (anzahlTreffer == zellorte.length) {

    ergebnis = "Versenkt";

  } // if-Ende

  System.out.println(ergebnis);

  return ergebnis;

} // Methodenende
```

Den String in ein int umwandeln.

Eine Variable erstellen, die das Ergebnis festhält, das wir zurückliefern möchten. Als Default-Wert >>Vorbei<< einsetzen (d.h., wir gehen von >>Vorbei<< aus).

Mit jeder Zelle im Array zellorte wiederholen (jedem Zellort im Objekt).

Dieses Array-Element (Zelle) mit dem Benutzervorschlag vergleichen.

Wenn wir einen Treffer haben!

Aus der Schleife ausbrechen, das Prüfen der anderen Zellen ist überflüssig.

Wir haben die Schleife verlassen, prüfen, ob wir jetzt >>versenkt<< sind (dreimal getroffen wurden), und ändern das String-Ergebnis in >>Versenkt<<.

Dem Benutzer das Ergebnis anzeigen (>>Vorbei<<, wenn es nicht in >>Treffer<< oder >>Versenkt<< geändert wurde).

Das Ergebnis an die aufrufende Methode zurückliefern.

Vorcode | Testcode | Hauptcode

## Nur die neuen Sachen

Die Dinge, die wir bisher noch nicht gesehen haben, finden Sie auf dieser Seite. Hören Sie auf, sich den Kopf zu zerbrechen! Die restlichen Details gibt es am Ende dieses Kapitels. Das hier soll Sie nur am Laufen halten.

### 1 Einen String in ein int umwandeln

Eine Klasse, die mit Java ausgeliefert wird.

Eine Methode in der Klasse Integer, die weiß, wie man einen String zu dem int >>parst<<, das er repräsentiert.

Erwartet einen String.

```
Integer.parseInt("3")
```

### 2 Die for-Schleife

Lesen Sie diese Deklaration einer for-Schleife als >>Wiederhole für jedes Element im Array >zellorte<: Nimm das nächste Element im Array und weise es der int-Variablen >zelle< zu.<<

Der Doppelpunkt (:) heißt >>in<<. Das ganze Ding bedeutet also >>für jeden int-Wert IN zellorte ...<<

```
for (int zelle : zellorte) { }
```

Deklariert eine Variable, die ein Element aus dem Array festhält. Bei jedem Schleifendurchlauf enthält diese Variable (hier die int-Variable namens >>zelle<<) ein anderes Element aus dem Array, bis es keine Elemente mehr gibt (oder der Code ein >>break<< ausführt ... siehe Punkt 4 weiter unten).

Das Array, über das in der Schleife iteriert wird. Bei jedem Durchlauf der Schleife wird der Variablen >>zelle<< das nächste Element des Arrays zugewiesen. (Mehr dazu erfahren Sie am Ende dieses Kapitels.)

### 3 Der Post-Inkrement-Operator

Das ++ heißt, addiere zu dem, was bereits da ist, 1 hinzu (oder, anders gesagt, erhöhe/inkrementiere um 1).

```
anzahlTreffer++
```

anzahlTreffer++ besagt (in diesem Fall) das Gleiche wie anzahlTreffer = anzahlTreffer + 1, ist aber etwas effizienter.

### 4 break-Anweisung

```
break;
```

Verlässt eine Schleife. Sofort. Direkt hier. Keine Iteration mehr, kein Boolescher Test mehr, nur raus, und zwar jetzt!

Vorcode | Testcode | Hauptcode

## Es gibt keine Dummen Fragen

**F: Was passiert in Integer.parseInt(), wenn das Ding, das man übergibt, keine Zahl ist? Und erkennt die Methode ausgeschriebene Zahlen wie »drei«?**

A: Integer.parseInt() funktioniert nur für Strings, die ASCII-Werte für Ziffern (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) repräsentieren. Wenn Sie versuchen, etwas wie »zwei« oder »blub« zu parsen, geht der Code zur Laufzeit hoch. (Mit *Hochgehen* meinen wir eigentlich *eine Exception auslösen*. Aber über Exceptions reden wir erst im Exception-Kapitel. Im Augenblick kommt *Hochgehen* der Sache nah genug.)

**F: Im Anfang des Buchs gab es ein Beispiel für eine *for*-Schleife, das ganz anders war als dieses hier – gibt es zwei verschiedene Arten von *for*-Schleifen?**

A: Ja! Seit der ersten Java-Version hat es eine Art *for*-Schleife (die weiter unten in diesem Kapitel erklärt wird) gegeben, die folgendermaßen aussieht:

```
for (int i = 0; i < 10; i++) {
  // mach irgendwas 10 Mal
}
```

Sie können dieses Format für jede Schleife verwenden, die Sie benötigen. Aber ... seit Java 5.0 (Tiger) können Sie auch die *enhanced* for-Schleife verwenden (das ist die offizielle Beschreibung, in diesem Buch präferieren wir jedoch was Deutscheres, die *verbesserte* for-Schleife), wenn Sie die Elemente in einem Array (oder, wie Sie im *nächsten* Kapitel sehen werden, einem *anderen* Typ von Collection) verwenden. Sie können immer die gute alte for-Schleife verwenden, aber die *verbesserte* for-Schleife vereinfacht Ihnen die Arbeit.

## Der endgültige Code für EinfachesDotCom und EinfachesDotComTestlauf

```
public class EinfachesDotComTestlauf {

   public static void main (String[] args) {
      EinfachesDotCom dot = new EinfachesDotCom();
      int[] orte = {2,3,4};
      dot.setZellorte(orte);
      String tipp = "2";
      String ergebnis = dot.prüfDich(tipp);
   }
}
```

---

```
public class EinfachesDotCom {

   int[] zellorte;
   int anzahlTreffer;

   public void setZellorte(int[] orte) {
      zellorte = orte;
   }

   public String prüfDich(String stringTipp) {
      int tipp = Integer.parseInt(stringTipp);
      String ergebnis = "Vorbei";
      for (int zelle : zellorte) {
         if (tipp == zelle) {
            ergebnis = "Treffer";
            anzahlTreffer++;
            break;
         } // if-Ende
      } // for-Ende

      if (anzahlTreffer ==
          zellorte.length) {
          ergebnis = "Versenkt";
      } // if-Ende
      System.out.println(ergebnis);
         return ergebnis;
   } // Methodenende
} // Klassenende
```

Hier verbirgt sich ein kleiner Bug. Der Code lässt sich kompilieren, aber manchmal ... zerbrechen Sie sich im Augenblick darüber nicht den Kopf, aber später werden wir uns der Sache stellen *müssen*.

**Was sollten wir sehen, wenn wir diesen Code ausführen?**

Der Testcode erstellt ein EinfachesDotCom-Objekt und gibt ihm einen Ort bei 2, 3, 4. Dann sendet er einen vorgetäuschten Rateversuch mit dem Wert »2« an die Methode prüfDich(). Wenn der Code richtig funktioniert, sollten wir die folgende Ausgabe sehen:

```
java EinfachesDotComTestlauf
Treffer
```

Vorcode | Testcode | Hauptcode

## Spitzen Sie Ihren Bleistift

Wir haben eine Testklasse und die Klasse EinfachesDotCom erstellt. Aber das eigentliche *Spiel* haben wir noch nicht erstellt. Nehmen Sie den Code auf der gegenüberliegenden Seite und die Spezifikation für das Spiel und schreiben Sie Ihre Ideen für den Vorcode für die Spielklasse auf. Wir haben Ihnen ein paar Zeilen vorgegeben, um Ihnen einen Ansatz zur Verfügung zu stellen. Der eigentliche Spielcode befindet sich auf der nächsten Seite, ***blättern Sie also nicht um, bevor Sie diese Übung gemacht haben!***

Sie sollten zwischen 12 und 18 Zeilen haben (einschließlich der Zeilen, die wir geschrieben haben, aber *ausschließlich* der Zeilen, in denen nur geschweifte Klammern stehen).

**METHODE** *public static void main (String [] args)*

**DEKLARATION**: Eine int-Variable, die die Anzahl der Rateversuche des Benutzers aufnimmt, mit dem Namen *anzahlTipps*.

**BERECHNE** eine Zufallszahl zwischen 0 und 4, die als Position für die erste Zelle dient.

**SOLANGE** das DotCom lebt:

  **HOLE** Benutzereingabe von der Kommandozeile.

**Das EinfacheDotComSpiel muss Folgendes machen:**

1. Das EinfachesDotCom-Objekt erstellen.

2. Einen Ort für das Objekt generieren (drei aufeinander folgende Zellen in einer einzelnen Zeile mit sieben virtuellen Zellen).

3. Den Benutzer zur Eingabe eines Rateversuchs auffordern.

4. Den Rateversuch prüfen.

5. Wiederholen, bis das DotCom versenkt ist.

6. Dem Benutzer sagen, wie viele Rateversuche er benötigt hat.

### Eine vollständige Spielinteraktion

```
Datei Bearbeiten Fenster Hilfe Weglaufen
%java EinfachesDotComSpiel
Geben Sie eine Zahl ein 2
Treffer
Geben Sie eine Zahl ein 3
Treffer
Geben Sie eine Zahl ein 4
Vorbei
Geben Sie eine Zahl ein 1
Versenkt
Sie haben 4 Rateversuche
benötigt
```

## Vorcode für die Klasse EinfachesDotComSpiel

### Alles passiert in main()

Es gibt da einige Dinge, die Sie einfach glauben müssen. Beispielsweise gibt es im Vorcode eine Zeile, die sagt: »HOLE Benutzereingabe von der Kommandozeile.« Ich gebe gern zu, dass das etwas mehr ist, als wir hier aus dem Nichts implementieren wollen. Aber glücklicherweise verwenden wir OO. Und das bedeutet, dass Sie eine *andere* Klasse/ein *anderes* Objekt bitten können, etwas für Sie zu tun, ohne sich darum kümmern zu müssen, ***wie*** es das macht. Wenn Sie Vorcode schreiben, sollten Sie voraussetzen, dass Sie *irgendwie* schon in der Lage sein werden, alles zu tun, was Sie tun müssen. Dann können Sie Ihren Verstand ganz der Ausarbeitung der Logik widmen.

*public static void main (String [] args)*

**DEKLARATION**: Eine int-Variable, die die Anzahl der Rateversuche des Benutzers speichert, mit dem Namen *anzahlTipps* und dem Initialisierungswert 0.

**MACHE** eine neue EinfachesDotCom-Instanz.

**BERECHNE** eine Zufallszahl zwischen 0 und 4, die als Position der ersten Zelle dient.

**MACHE** ein int-Array mit 3 ints, das die generierte Zufallszahl, die Zufallszahl plus 1 und die Zufallszahl plus 2 enthält (Beispiel: 3,4,5).

**RUFE** die Methode *setZellorte()* auf der EinfachesDotCom-Instanz auf.

**DEKLARATION**: Eine boolean-Variable, die den Zustand des Spiels repräsentiert, mit dem Namen *lebt*.
**SETZE** sie auf wahr.

**SOLANGE** das Dot-Com lebt (lebt == true):
  **HOLE** Benutzereingabe von der Kommandozeile.
  ***// PRÜFE*** den Rateversuch des Benutzers.
  **RUFE** die Methode *prüfDich()* auf der EinfachesDotCom-Instanz auf.
  **INKREMENTIERE** die Variable *anzahlTipps*.
  ***// PRÜFE,*** ob Dot-Com versenkt ist.
  **WENN** Ergebnis »Versenkt« ist:
    **SETZE** *lebt* auf false (was bewirkt, dass die Schleife nicht noch einmal betreten wird).
    **GIB** die Anzahl von Rateversuchen aus.
  WENN-ENDE
SOLANGE-ENDE
METHODENENDE

**Lerntipp**

Arbeiten Sie nicht zu lange einseitig mit einer Hälfte Ihres Gehirns. Wenn Sie 30 Minuten lang nur mit der linken Hälfte Ihres Gehirns arbeiten, ist das, als würden Sie 30 Minuten lang nur mit Ihrem linken *Arm* arbeiten. Geben Sie den Hälften Ihres Gehirns Pausen, indem Sie regelmäßig die Seiten wechseln. Wenn Sie zu einer Seite wechseln, kann sich die andere Seite ausruhen und erholen. Die linke Hälfte des Gehirns ist bei Aufgaben wie schrittweisen Sequenzen, der Lösung logischer Probleme und der Problemanalyse aktiv. Die rechte Hälfte springt bei Metaphern, kreativer Problemlösung, Mustervergleichen und Visualisierungsprozessen ein.

## Punkt für Punkt

- Ausgangspunkt Ihres Java-Programms sollte ein abstrakter Entwurf sein.
- Normalerweise schreiben Sie drei Dinge, wenn Sie eine neue Klasse schreiben:

  ***Vorcode***

  ***Testcode***

  ***Hauptcode (Java)***

- Vorcode sollte beschreiben, *was* zu tun ist, nicht *wie* es zu tun ist. Die Implementierung kommt später.
- Nutzen Sie den Vorcode als Hilfe beim Entwurf des Testcodes.
- Schreiben Sie den Testcode, *bevor* Sie die Methoden implementieren.
- Verwenden Sie *for*-Schleifen statt *while*-Schleifen, wenn Sie wissen, wie oft Sie den Schleifencode wiederholen wollen.
- Verwenden Sie den Prä-/Post-*Inkrement*-Operator, um einer Variablen 1 hinzuzufügen (x++;).
- Verwenden Sie den Prä-/Post-*Dekrement*-Operator, um von einer Variablen 1 abzuziehen (x--;).
- Verwenden Sie `Integer.parseInt()`, um den int-Wert eines Strings zu erhalten.
- `Integer.parseInt()` funktioniert nur bei Strings, die eine Ziffer (0, 1, 2 usw.) repräsentieren.
- Nutzen Sie *break*, um eine Schleife vorzeitig zu verlassen (selbst wenn die Boolesche Testbedingung immer noch wahr ist).

Vorcode | Testcode | Hauptcode

# Die main()-Methode des Spiels

Denken Sie wie bei der Klasse EinfachesDotCom darüber nach, welche Teile dieses Codes Sie verbessern möchten (oder müssen). Die nummerierten Dinger (1) stehen für etwas, das wir hervorheben möchten. Sie werden auf der gegenüberliegenden Seite erläutert. Und falls Sie sich fragen, warum wir bei dieser Klasse die Testcode-Phase übersprungen haben ... für das Spiel brauchen wir keine Testklasse. Es hat nur eine Methode, was sollten wir also im Testcode testen? Eine *separate* Klasse erstellen, die *main()* auf *dieser* Klasse aufruft? Das haben wir uns erspart.

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

    int anzahlTipps = 0;

    SpielHelfer helfer = new SpielHelfer();

    EinfachesDotCom dasDotCom = new EinfachesDotCom();

    int zufallsZahl = (int) (Math.random() * 5);

    int[] orte = {zufallsZahl, zufallsZahl+1, zufallsZahl+2};

    dasDotCom.setZellorte(orte);

    boolean lebt = true;

    while(lebt == true) {

       String tipp = helfer.getBenutzereingabe("Geben Sie eine Zahl ein");

       String ergebnis = dasDotCom.prüfDich(tipp);

       anzahlTipps++;

       if (ergebnis.equals("Versenkt")) {

          lebt = false;

          System.out.println("Sie haben " + anzahlTipps + " Versuche benötigt");

       } // if-Ende

    } // while-Ende

} // main-Ende
```

Eine Variable erzeugen, um die Anzahl der Versuche des Benutzers nachzuhalten.

Das ist eine besondere Klasse, die wir geschrieben haben. Sie hat eine Methode, die uns die Benutzereingabe verschafft. Nehmen Sie für den Augenblick einfach an, dass sie ein Teil von Java ist.

Das DotCom-Objekt erzeugen.

Die Zufallszahl für die erste Zelle generieren und verwenden, um das Zellorte-Array einzurichten.

Dem DotCom seinen Ort (das Array) geben.

Eine boolean-Variable anlegen, die nachhält, ob das Spiel noch lebt, und im Test der while-Schleife verwendet wird. Wiederholen, solange das Spiel lebt.

Den String mit der Benutzereingabe abrufen.

Das DotCom zur Prüfung der Eingabe auffordern und das zurückgelieferte Ergebnis in einem String speichern.

Die Anzahl der Rateversuche erhöhen.

Ist das DotCom »Versenkt«? Wenn ja, lebt auf false setzen (damit wir nicht wieder in die Schleife eintreten) und die Anzahl der Rateversuche ausgeben.

**DEKLARATION**: Eine int-Variable, die die Anzahl der Rateversuche des Benutzers speichert, mit dem Initialisierungswert 0.

**MACHE** eine neue EinfachesDotCom-Instanz.

**BERECHNE** eine Zufallszahl zwischen 0 und 4.

**MACHE** ein int-Array mit den 3 Zellorten.

**RUFE** *setZellorte()* auf der EinfachesDotCom-Instanz auf.

**DEKLARATION**: Eine boolean-Variable mit dem Namen *lebt*.

**SOLANGE** das Dot-Com lebt,

**HOLE** Benutzereingabe von der Kommandozeile.

*// PRÜFE* sie.

**RUFE** die Methode *prüfDich()* auf der EinfachesDotCom-Instanz auf.

**INKREMENTIERE** *anzahlTipps*.

**WENN** Ergebnis »Versenkt« ist

**SETZE** *lebt* auf false.

**GIB** die Anzahl von Rateversuchen aus.

Vorcode | Testcode | Hauptcode

## random() and getBenutzereingabe()

Zwei Dinge, bei denen noch ein wenig Erläuterungsbedarf besteht, finden Sie auf dieser Seite. Das ist nur ein kurzer Überblick, damit Sie weitermachen können. Mehr Details zur Klasse SpielHelfer finden Sie auf der folgenden Seite.

1. Eine Zufallszahl generieren

Das ist ein Cast (eine Typumwandlung), der das unmittelbar darauf folgende Ding dazu zwingt, den Typ des Casts (d.h. den in den Klammern angegebenen Typ) anzunehmen. Math.random liefert einen double-Wert zurück, den wir also erst zu einem int (wir möchten eine nette ganze Zahl zwischen 0 und 4) casten müssen. In diesem Fall schneidet der Cast den Nachkommateil des double-Werts ab.

Die Methode Math.random liefert eine Zahl zwischen null und irgendetwas minimal kleiner als eins zurück. Also liefert diese Formel (mit dem Cast) eine Zahl zwischen 0 und 4 (d.h. 0-4,999... auf ein int gecastet) zurück.

```
int zufallsZahl = (int) (Math.random() * 5)
```

Wir deklarieren eine int-Variable, die die Zufallszahl festhalten soll, die wir zurückerhalten.

Eine Klasse, die mit Java ausgeliefert wird.

Eine Methode der Klasse Math.

2. Mit der Klasse SpielHelfer die Benutzereingabe abfragen

Eine zuvor erzeugte Instanz einer Klasse, die wir als Unterstützungsklasse für das Spiel geschrieben haben. Sie wird SpielHelfer genannt, und Sie haben sie bisher nicht gesehen (das kommt noch).

Diese Methode erwartet ein String-Argument, das verwendet wird, um den Benutzer auf der Kommandozeile zu einer Eingabe aufzufordern. Das, was Sie hier übergeben, wird auf dem Terminal angezeigt, bevor die Methode beginnt, auf die Benutzereingabe zu warten.

```
String tipp = helfer.getBenutzereingabe("Geben Sie eine Zahl ein");
```

Wir deklarieren eine String-Variable, die den String mit der Benutzereingabe festhalten soll, den wir zurückerhalten (»3«, »5« usw.).

Eine Methode der Klasse SpielHelfer, die den Benutzer auf der Kommandozeile zu einer Eingabe auffordert und diese einliest, nachdem der Benutzer ENTER gedrückt hat und das Ergebnis als String zurückliefert.

# Eine letzte Klasse: SpielHelfer

**Wir haben die *DotCom*-Klasse gemacht.**

**Wir haben die *Spiel*-Klasse gemacht.**

**Jetzt müssen wir uns nur noch um die *Helfer*-Klasse kümmern** – die mit der Methode getBenutzereingabe(). Der Code zum Einlesen der Kommandozeile geht über das hinaus, was wir im Augenblick erklären möchten. Er wirft zu viele Fragen auf, die wir uns besser für später aufsparen.

(Später, beispielsweise in Kapitel 14.)

Kopieren* Sie einfach den unten stehenden Code und kompilieren Sie ihn zu einer Klasse namens SpielHelfer. Packen Sie alle drei Klassen (EinfachesDotCom, EinfachesDotComSpiel und SpielHelfer) in das gleiche Verzeichnis und machen Sie es zu Ihrem Arbeitsverzeichnis.

Jedes Mal, wenn Sie auf das Logo

stoßen, sehen Sie Code, den Sie so eingeben müssen, wie er da steht, und einfach hinnehmen müssen. Vertrauen Sie uns! Wie dieser Code funktioniert, werden Sie *später* lernen.

Code-
Fertiggericht

```
import java.io.*;
public class SpielHelfer {
   public String getBenutzereingabe(String prompt) {
      String eingabeZeile = null;
      System.out.print(prompt + " ");
      try {
        BufferedReader is = new BufferedReader(
        new InputStreamReader(System.in));
        eingabeZeile = is.readLine();
        if (eingabeZeile.length() == 0 ) return null;
      } catch (IOException e) {
        System.out.println("IOException: " + e);
      }
      return eingabeZeile;
   }
}
```

*Wir wissen, wie viel Spaß Sie am Tippen haben. Aber für die seltenen Momente, in denen Sie lieber etwas anderes machen möchten, haben wir das Code-Fertiggericht auf der O'Reilly-Website zum Buch verfügbar gemacht.

## Lasst uns spielen

Hier sehen Sie, was passiert, wenn wir das Spiel starten und die Zahlen 1, 2, 3, 4, 5, 6 eingeben. Sieht gut aus.

### Ein vollständiger Spielablauf

(Ihre Ausgabe kann anders aussehen.)

```
Datei Bearbeiten Fenster Hilfe Lächeln
%java EinfachesDotComSpiel
Geben Sie eine Zahl ein 1
Vorbei
Geben Sie eine Zahl ein 2
Vorbei
Geben Sie eine Zahl ein 3
Vorbei
Geben Sie eine Zahl ein 4
Treffer
Geben Sie eine Zahl ein 5
Treffer
Geben Sie eine Zahl ein 6
Versenkt
Sie haben 6 Versuche benötigt
```

## Was ist das? Ein Bug?

### Schluck!

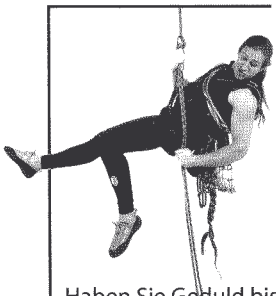
Hier sehen Sie, was passiert, wenn wir 3, 3, 3 eingeben.

### Ein anderer Spielablauf

(Autsch!!)

```
Datei Bearbeiten Fenster Hilfe Ohnmacht
%java EinfachesDotComSpiel
Geben Sie eine Zahl ein 3
Treffer
Geben Sie eine Zahl ein 3
Treffer
Geben Sie eine Zahl ein 3
Versenkt
Sie haben 3 Versuche benötigt
```

Spitzen Sie Ihren Bleistift



### Wir hängen über dem Abgrund!

Werden wir den Bug ***finden***?

Werden wir den Bug ***reparieren können***?

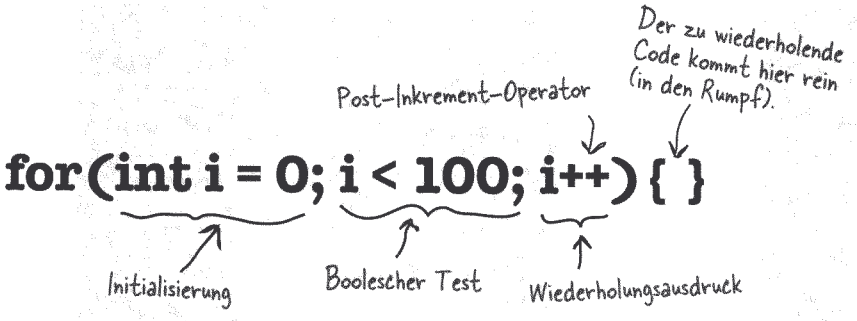
Haben Sie Geduld bis zum nächsten Kapitel, in dem wir diese Fragen beantworten werden und mehr ...

Und inzwischen versuchen Sie zu überlegen, was schief gelaufen ist und wie man es reparieren könnte.

# Mehr zu for-Schleifen

Wir haben jetzt den ganzen Spielcode für *dieses* Kapitel behandelt (aber wir werden ihn im nächsten Kapitel wieder anpacken, um die De-luxe-Version des Spiels zu schreiben). Wir wollten Ihre Arbeit nicht mit überflüssigen Details und Hintergrundinformationen unterbrechen, deswegen haben wir damit bis jetzt gewartet. Wir beginnen mit ausführlichen Informationen zu for-Schleifen, und wenn Sie C++-Programmierer sind, können Sie diese letzten paar Seiten einfach überspringen ...

## Normale (nicht verbesserte) for-Schleifen




**In normalem Deutsch bedeutet das:** »100 Mal wiederholen.«

**Wie der Compiler es sieht:**

* Erstelle eine Variable *i* und setze sie auf 0.
* Wiederhole, solange *i* kleiner 100 ist.
* Addiere am Ende jedes Schleifendurchlaufs 1 zu *i* hinzu.

### Teil eins: *Initialisierung*

Verwenden Sie diesen Teil, um eine Variable zu deklarieren und initialisieren, die im Schleifenrumpf verwendet wird. Am häufigsten werden Sie diese Variable als Zähler verwenden. Sie können hier auch mehrere Variablen initialisieren. Aber das werden wir uns später im Buch ansehen.

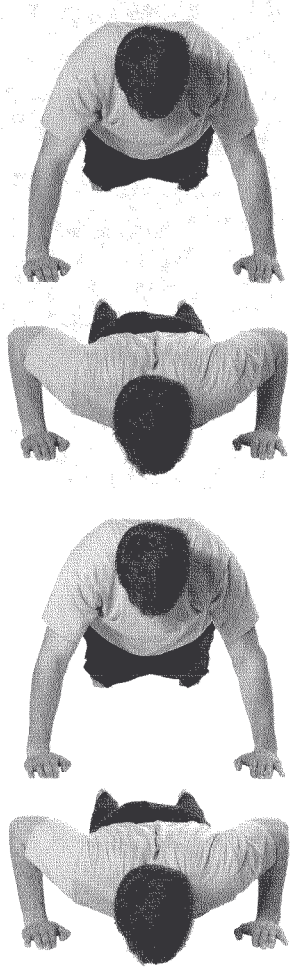
### Teil zwei: *Boolescher Test*

Hier kommt die Testbedingung rein. Egal was hier steht, es *muss* zu einem boolean-Wert aufgelöst werden (***true*** oder ***false***, wie Sie wissen). Sie können einen Test wie (x >= 4) angeben, aber Sie können auch eine Methode aufrufen, die einen Booleschen Wert zurückliefert.

### Teil drei: *Wiederholungsausdruck*

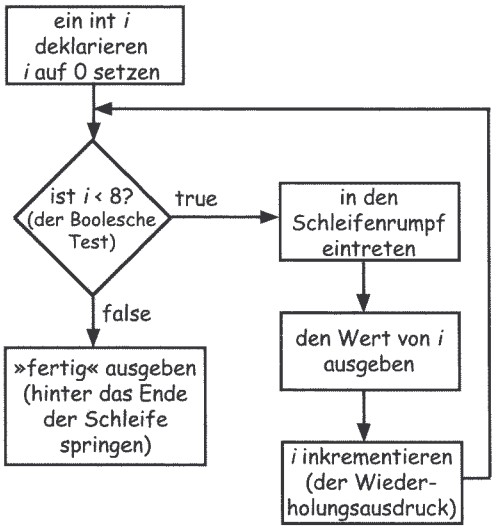
In diesem Teil geben Sie ein oder mehrere Dinge an, die bei jedem Schleifendurchlauf passieren sollen. Denken Sie daran, dass diese Dinge am *Ende* jedes Schleifendurchlaufs passieren.

**100 Mal wiederholen:**



# Eine Schleife durchreisen

```
for (int i = 0; i < 8; i++) {
   System.out.println(i);
}
System.out.println("fertig");
```




## Der Unterschied zwischen for und while

Eine *while*-Schleife hat nur einen Booleschen Test und keinen eingebauten Initialisierungs- und Wiederholungsausdruck. Eine *while*-Schleife ist geeignet, wenn Sie nicht wissen, wie oft die Schleife durchlaufen werden muss, und die Ausführung fortgesetzt werden soll, solange eine Bedingung true ist. Aber wenn Sie *wissen*, wie oft die Schleife durchlaufen werden soll (z.B. die Länge eines Arrays, siebenmal usw.), ist eine *for*-Schleife sauberer. Hier sehen Sie, wie man die Schleife von oben für *while* umschreiben müsste:

```
int i = 0;                    <- Wir müssen den Zähler deklarieren und initialisieren.
while (i < 8) {
   System.out.println(i);
   i++;                       <- Wir müssen den Zähler inkrementieren.
}
System.out.println("fertig");
```

**Ausgabe:**

```
Datei Bearbeiten Fenster Hilfe Wiederholen
%java Test
0
1
2
3
4
5
6
7
fertig
```

++ --

### Prä- und Post-Inkrement-/Dekrement-Operator

Die Kurzform, mit der einer Variablen 1 hinzugefügt wird:

```
x++;
```

bedeutet das Gleiche wie:

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

Beides meint in diesem Kontext dasselbe:
»Füge dem aktuellen Wert von x 1 hinzu« oder »***inkrementiere*** x um 1«.

Und die Kurzform, mit der von einer Variablen 1 abgezogen wird:

```
x--;
```

ist das Gleiche wie:

```
x = x - 1;
```

Natürlich ist das nie die ganze Geschichte. Die Stellung des Operators (entweder vor oder nach der Variablen) kann das Ergebnis beeinflussen. Wenn Sie den Operator *vor* der Variablen angeben (beispielsweise ++x), heißt das »inkrementiere *erst* um 1 und verwende *dann* diesen neuen Wert von x«. Das spielt nur eine Rolle, wenn ++x ein Teil eines größeren Ausdrucks und keine separate Anweisung ist.

```
int x = 0;      int z = ++x;
```

Das ergibt: x ist 1, z ist 1

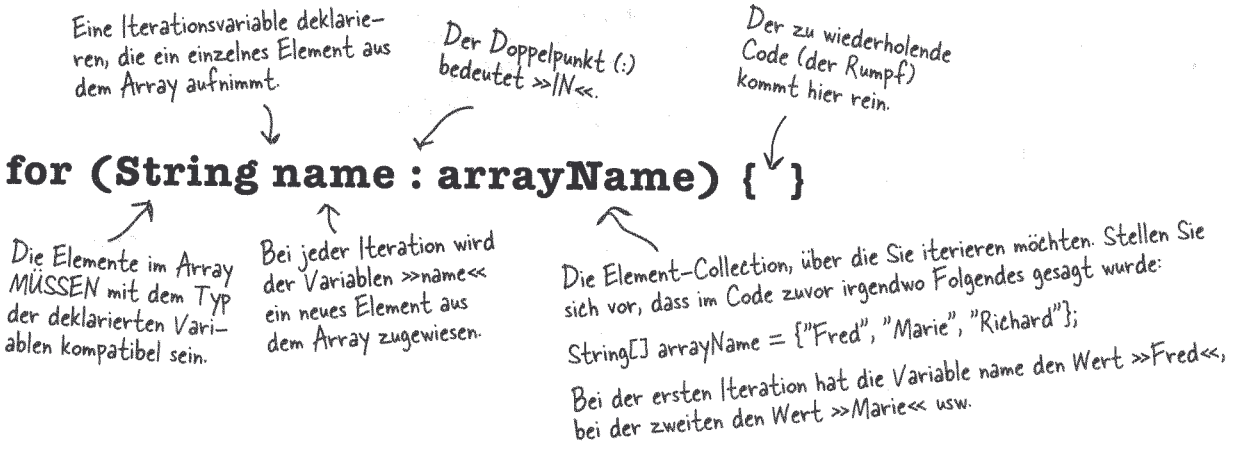
Aber wenn Sie das ++ *hinter* dem x platzieren, erhalten Sie ein anderes Ergebnis:

```
int x = 0;      int z = x++;
```

Das ergibt: x ist 1, aber ***z ist 0***! z erhält den Wert von x, und *dann erst* wird x inkrementiert.

# Die verbesserte for-Schleife

Seit Java 5.0 (Tiger) besitzt die Programmiersprache Java eine zweite Art von *for*-Schleife, die als *verbessertes for* bezeichnet wird und es leichter macht, über alle Elemente in einem Array oder in anderen Arten von Collections zu iterieren. (Über *andere* Collections werden Sie im nächsten Kapitel mehr erfahren.) Das ist eigentlich alles, was Ihnen das verbesserte for bietet – ein einfacheres Mittel, die Elemente in einer Collection zu durchlaufen. Aber da das die häufigste Verwendung einer *for*-Schleife ist, war es nützlich, das der Sprache hinzuzufügen. Wir werden uns die *verbesserte for-Schleife* im nächsten Kapitel erneut vornehmen, wenn wir über Collections reden, die *keine* Arrays sind.




**Was in normalem Deutsch bedeutet:** »Für jedes Element in arrayName weise das Element der Variablen ›name‹ zu und führe dann den Schleifenrumpf aus.«

**Wie der Compiler es sieht:**

* Erzeuge eine String-Variable namens *name* und setze sie auf null.
* Weise *name* den ersten Wert in *arrayName* zu.
* Führe den Schleifenrumpf (den Codeblock, der von den geschweiften Klammern eingeschlossen wird) aus.
* Weise *name* den nächsten Wert in *arrayName* zu.
* Wiederhole den Vorgang, *solange noch Elemente im Array vorhanden sind.*

Hinweis: Je nach der Programmiersprache, die sie in der Vergangenheit verwendet haben, bezeichnen einige Programmierer die verbesserte for-Schleife als >>for each<<- oder>>for in<<-Schleife, weil sie genau so gelesen werden kann: >>for EACH Ding IN der Collection ...<<

**Teil eins: *Deklaration der Iterationsvariablen***

Verwenden Sie diesen Teil, um die Variable zu deklarieren und zu initialisieren, die im Schleifenrumpf verwendet werden soll. Bei jedem Durchlauf der Schleife enthält die Variable ein anderes Element aus der Collection. Der Typ dieser Variablen muss mit dem Typ der Elemente der Collection kompatibel sein! Beispielsweise können Sie keine *int*-Iterationsvariable deklarieren, die mit einem *String[]*-Array verwendet werden soll.

**Teil zwei: *Die eigentliche Collection***

Das muss eine Referenz auf ein Array oder eine andere Collection sein. Noch mal – machen Sie sich keine Gedanken über *andere* nicht-Array-artige Collections – diese werden Sie im nächsten Kapitel kennen lernen.

### Einen String in ein int umwandeln

```
int tipp = Integer.parseInt(stringTipp);
```

Der Benutzer gibt seinen Rateversuch auf der Kommandozeile ein, wenn das Programm ihn dazu auffordert. Dieser Rateversuch kommt als String ("2","0" usw.) an, und das Spiel reicht den String an die Methode prüfDich() weiter.

Aber Zellorte sind einfach ints in einem Array, die man nicht mit einem String vergleichen kann.

Folgendes ***würde*** beispielsweise ***nicht funktionieren:***

String zahl = "2";

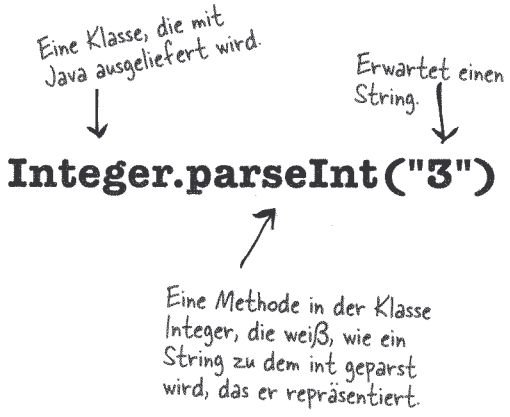
int x = 2;

if (x == zahl) // schreckliche Explosion!

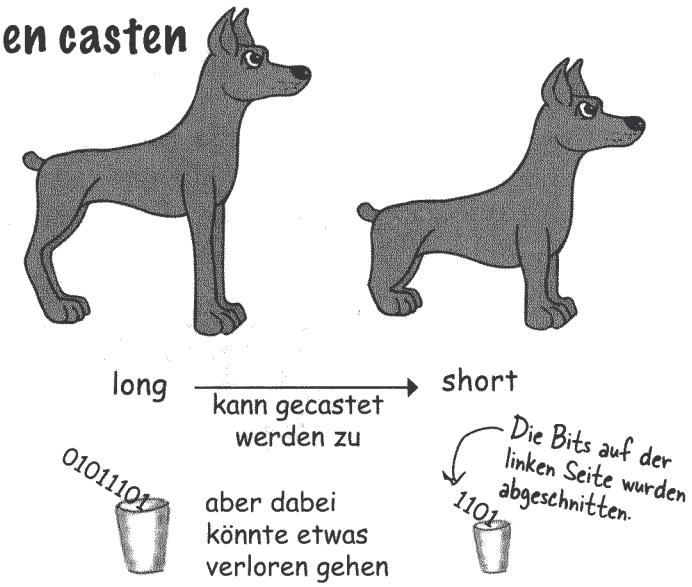
Wenn Sie das versuchen, lacht der Compiler und macht sich über Sie lustig:

```
operator == cannot be applied to
      int,java.lang.String
    if (x == zahl) { }
           ^
```

Um diese Apfel-und-Birnen-Geschichte zu umgehen, müssen wir den *String* "2" zu dem *int* 2 machen. In die Java-Klassenbibliothek ist eine Klasse namens Integer (richtig, eine Integer-*Klasse*, nicht der *elementare* Typ int) eingebaut, und eine ihrer Aufgaben ist es, Strings, die Zahlen *repräsentieren*, zu nehmen und in *richtige* Zahlen umzuwandeln.




## Elementare Typen casten




In Kapitel 3 haben wir über die Größen der verschiedenen elementaren Typen gesprochen und dass Sie etwas Großes nicht direkt in etwas Kleines stopfen können:

```
long y = 42;
int x = y; // lässt sich nicht kompilieren
```

Ein *long* ist größer als ein *int*, und der Compiler kann nicht wissen, wo sich das *long* herumgetrieben hat. Vielleicht war es mit den anderen longs einen trinken und hat über die richtig großen Werte gesprochen. Um den Compiler zu zwingen, den Wert aus einer größeren elementaren Variablen in eine kleinere zu stecken, können Sie den **Cast**-Operator verwenden. Der sieht so aus:

```
long y = 42;      // bisher kein Problem
int x = (int) y; // x = 42 cool!
```

Indem Sie den Cast einbauen, sagen Sie dem Compiler, dass er den Wert von y nehmen, auf int-Größe zurechtschneiden und x dann auf den Wert setzen soll, der übrig bleibt. Wenn der Wert von y größer als der maximale Wert für x war, ist das, was übrig bleibt, eine seltsame (aber berechenbare*) Zahl:

```
long y = 40002;
// 40002 übersteigt die 16-Bit-Grenze eines short
short x = (short) y; // x ist jetzt -25534!
```

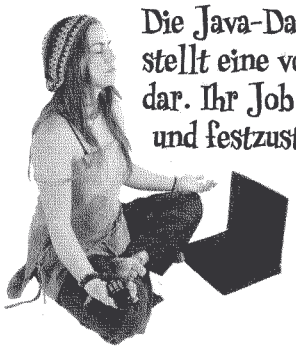
Der Punkt bleibt aber, dass der Compiler Ihnen erlaubt, das zu tun. Oder nehmen wir an, Sie haben eine Gleitkommazahl und möchten nur den ganzzahligen (*int*) Anteil davon haben:

```
float f = 3.14f;
int x = (int) f; // x ist jetzt 3
```

Und *denken* Sie nicht einmal daran, irgendetwas in einen Booleschen Wert zu verwandeln oder umgekehrt – lassen Sie einfach gleich die Finger davon.

**Das schließt Vorzeichenbit, binäre Zweierkomplemente und andere Verrücktheiten ein, die wir alle am Anfang von Anhang B besprechen werden.*

# Spielen Sie JVM



Die Java-Datei auf dieser Seite stellt eine vollständige Quelldatei dar. Ihr Job ist es, JVM zu spielen und festzustellen, was die Ausgabe wäre, wenn das Programm ausgeführt würde.

```
class Ausgabe {

  public static void main(String [] args) {
    Ausgabe a = new Ausgabe();
    a.los();
  }

  void los() {
    int y = 7;
    for(int x = 1; x < 8; x++) {
      y++;
      if (x > 4) {
        System.out.print(++y + " ");
      }
      if (y > 14) {
        System.out.println(" x = " + x);
        break;
      }
    }
  }
}
```

Datei Bearbeiten Fenster Hilfe OM

```
% java Ausgabe
12 14
```

-oder-

Datei Bearbeiten Fenster Hilfe Weihrauch

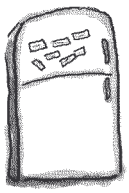
```
% java Ausgabe
12 14 x = 6
```

-oder-

Datei Bearbeiten Fenster Hilfe Glaube

```
% java Ausgabe
13 15 x = 6
```

# Code-Magneten



Ein funktionierendes Java-Programm ist vollkommen durcheinander geraten. 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 hinzu, wie Sie benötigen!

```
x++;
```

```
if (x == 1) {
```

```
System.out.println(x + " " + y);
```

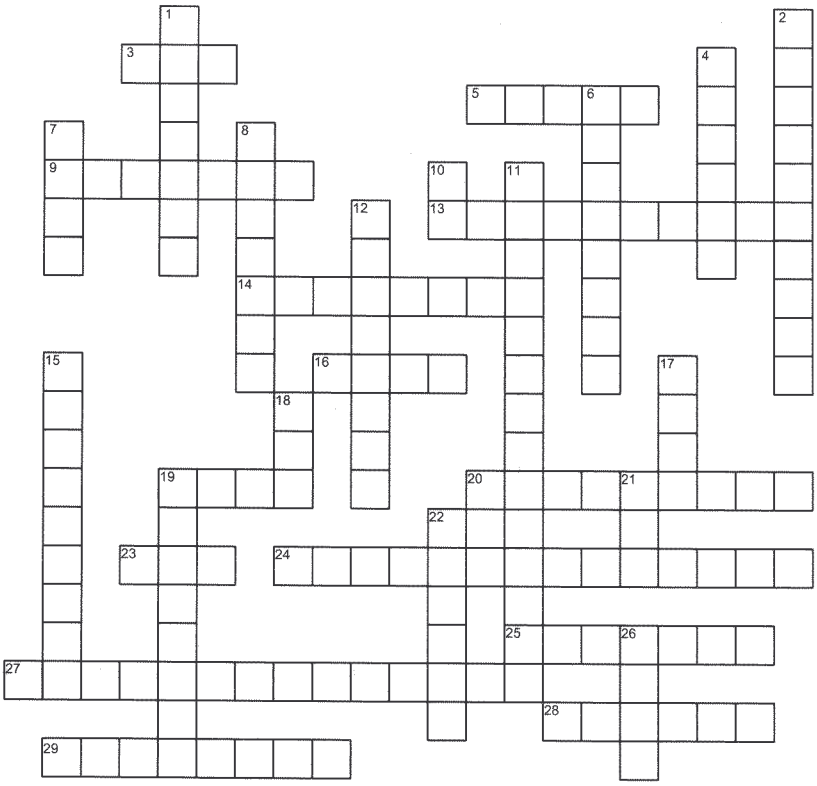
```
class MultiFor {
```

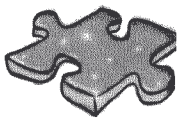
```
for(int  y = 4; y > 2; y--) {
```

```
for(int x = 0; x < 4; x++) {
```

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

```
Datei Bearbeiten Fenster Hilfe Überfall
% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```





# JavaKreuzworträtsel

Wie Ihnen ein Kreuzworträtsel helfen soll, Java zu lernen? Na, alle Wörter **beziehen** sich auf Java. Außerdem enthalten die Hinweise Metaphern, Wortspiele oder Ähnliches. Diese mentalen Knoten und Windungen brennen alternative Wege zu Java-Wissen direkt in Ihr Gehirn!

## Waagerecht

3 Mehrteilige Schleife.
5 Vorzeitig verlassen.
9 Sieht aus wie ein elementarer Typ, aber ...
13 Vorcode-artiger Code.
14 Prä-Inkrement ist ein Typ von _____.
16 Standard-Package-Namen, erster Teil.
19 Inkrementierungstyp.
20 Ein Durchlauf.
23 Methodenbotschafter (Kurzform).
24 Schickes Compusprech-Wort für einbauen.
25 Eine Array-Einheit.
27 Einen ersten Wert festlegen.
28 Methode zur Veränderung von Instanzvariablen.
29 Als ob ...

## Senkrecht

1 Lässt sich nicht casten.
2 Kompilieren und ____.
4 Math-Methode.
6 For besitzt einen Wiederholungs_____.
7 ++-Betrag.
8 Arbeitspferd von Klassen.
10 Propagiert die Philosophie: Tests zuerst!
11 Der Vernichtung entgegen.
12 Instanz oder lokal.
15 While oder For.
17 Umwandlung.
18 32-Bit-Typ.
19 Umwandlungsmethode.
21 Der große Werkzeugkasten.
22 Methodenantwort.
26 Pi-Haus.

## Vermischte Nachrichten

Unten ist ein kurzes Java-Programm aufgeführt. Ein Block des Programms fehlt. Ihre Aufgabe ist es, **die möglichen Codeblöcke** (links) **den Ausgaben** (rechts) **zuzuordnen**, die Sie sehen würden, wenn der Block eingefügt wurde. Nicht alle Zeilen mit Ausgabe werden verwendet, und einige Zeilen mit Ausgaben können mehrfach verwendet werden. Ziehen Sie Linien, die die Codeblock-Kandidaten mit den passenden Kommandozeilenausgaben verbinden.

```
class MischMasch5 {
  public static void main(String [] args) {
    int x = 0;
    int y = 30;
    for (int außen = 0; außen < 3; außen++) {
      for(int innen = 4; innen > 1; innen--) {

        [                                    ]  <-- Hier kommen die möglichen Codekandidaten hin.

        y = y - 2;
        if (x == 6) {
          break;
        }
        x = x + 3;
      }
      y = y - 2;
    }
    System.out.println(x + " " + y);
  }
}
```

**Kandidaten:**

```
x = x + 3;
x = x + 6;
x = x + 2;
x++;
x--;
x = x + 0;
```

**Mögliche Ausgaben:**

```
45 6
36 6
54 6
60 10
18 6
6 14
12 14
```

Ordnen Sie jedem Kandidaten eine der möglichen Ausgaben zu.

# Lösungen zu den Übungen

## Spielen Sie JVM:

```
class Ausgabe {

  public static void main(String [] args) {
    Ausgabe a = new Ausgabe();
    a.los();
  }
  void los() {
    int y = 7;
    for(int x = 1; x < 8; x++) {
      y++;
      if (x > 4) {
         System.out.print(++y + " ");
      }
      if (y > 14) {
        System.out.println(" x = " + x);
        break;
      }
    }
  }
}
```

Haben Sie daran gedacht, die break-Anweisung zu berücksichtigen? Welche Auswirkungen hatte das auf die Ausgabe?

```
Datei Bearbeiten Fenster Hilfe MotorradPflege

% java Ausgabe
13 15 x = 6
```

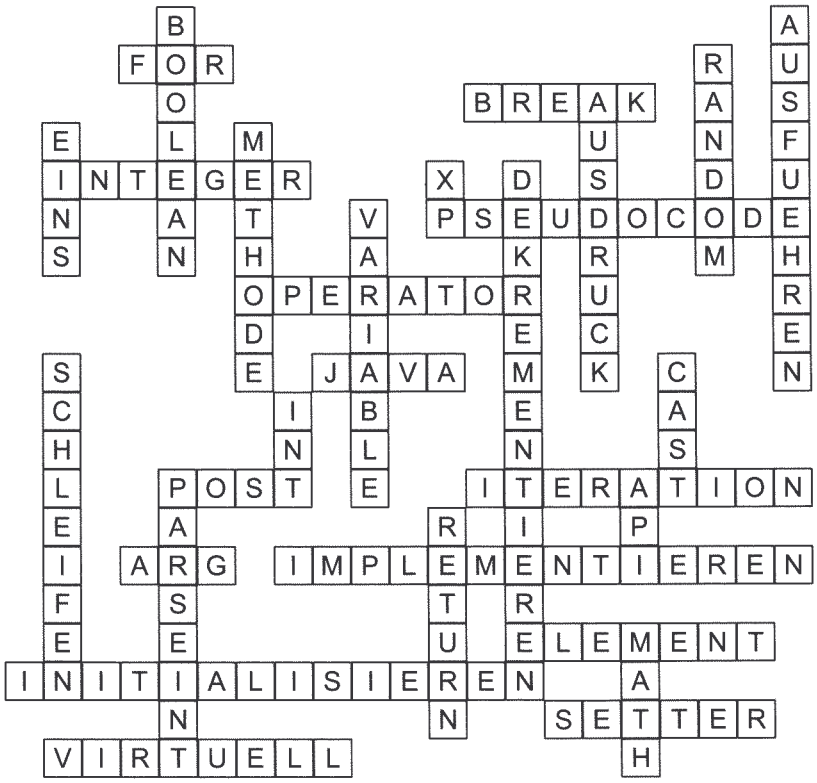
## Code-Magneten:

```
class MultiFor {

  public static void main(String [] args) {

    for(int x = 0; x < 4; x++) {

      for(int y = 4; y > 2; y--) {
        System.out.println(x + " " + y);
      }

      if (x == 1) {
        x++;
      }
    }
  }
}
```

Was würde passieren, wenn dieser Codeblock vor der »y-for-Schleife« stände?

```
Datei Bearbeiten Fenster Hilfe Monopol

% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```

## Puzzle-Lösungen




**Kandidaten:**

```
x = x + 3;
x = x + 6;
x = x + 2;
x++;
x--;
x = x + 0;
```

**Mögliche Ausgaben:**

```
45 6
36 6
54 6
60 10
18 6
6 14
12 14
```

# Die Java-Bibliothek verwenden

**Java wird mit Hunderten vorgefertigter Klassen ausgeliefert.** Sie müssen das Rad nicht jedes Mal neu erfinden, wenn Sie wissen, wie Sie das, was Sie brauchen, in der Java-Bibliothek finden, die als die **Java-API** bezeichnet wird. *Schließlich haben Sie etwas Besseres zu tun.* Wenn Sie Code schreiben, können Sie sich ruhig auf die Teile beschränken, die bei Ihrer Anwendung besonders sind. Kennen Sie die Programmierer, die jeden Nachmittag pünktlich um 5 durch die Tür sind? Die, die dafür morgens nicht vor 10 antanzen? **Sie nutzen die Java-API.** Und in ungefähr zehn Seiten werden auch Sie das tun. Die Java-Kernbibliothek ist ein gigantischer Haufen von Klassen, die nur darauf warten, dass Sie sie als Bausteine einsetzen, um Ihr Programm aus größtenteils vorgefertigtem Code zusammenzubauen. Die Java-Fertiggerichte, die wir in diesem Buch verwenden, bestehen aus Code, den Sie nicht aus dem Nichts zaubern müssen. Aber Sie müssen ihn immer noch eintippen. Die Java-API ist voll Code, den Sie nicht einmal *eintippen* müssen. Sie müssen nur lernen, wie man ihn verwendet.

# Im letzten Kapitel haben wir Sie über einem Abgrund - mit einem Bug - hängen lassen.

## Wie es aussehen sollte

Hier sehen Sie, was passiert, wenn wir das Spiel starten und die Zahlen 1, 2, 3, 4, 5, 6 eingeben. Sieht gut aus.

### Ein kompletter Spielablauf

(Ihr Ergebnis kann anders aussehen.)

```
Datei Bearbeiten Fenster Hilfe Lächeln
%java EinfachesDotComSpiel
Geben Sie eine Zahl ein 1
Vorbei
Geben Sie eine Zahl ein 2
Vorbei
Geben Sie eine Zahl ein 3
Vorbei
Geben Sie eine Zahl ein 4
Treffer
Geben Sie eine Zahl ein 5
Treffer
Geben Sie eine Zahl ein 6
Versenkt
Sie haben 6 Versuche benötigt
```

## Wie der Bug aussieht

Hier sehen Sie, was passiert, wenn wir 3, 3, 3 eingeben.

### Ein anderer Spielablauf

(Autsch)

```
Datei Bearbeiten Fenster Hilfe Ohnmacht
%java EinfachesDotComSpiel
Geben Sie eine Zahl ein 3
Treffer
Geben Sie eine Zahl ein 3
Treffer
Geben Sie eine Zahl ein 3
Versenkt
Sie haben 3 Versuche benötigt
```

**In der aktuellen Version müssen Sie einen bereits erzielten Treffer nur noch zweimal wiederholen, um ein Dot-Com zu versenken!**

# Was ist passiert?

Folgendes ist schief gelaufen. Wir haben immer dann einen Treffer gezählt, wenn der Benutzer einen Zellort richtig erraten hat – *auch dann, wenn der Ort schon getroffen worden ist!*

Wir brauchen eine Möglichkeit zu ermitteln, ob der Benutzer die Zelle, die er gerade getroffen hat, zuvor schon einmal getroffen hat. Hat er die Zelle schon einmal getroffen, dürfen wir das nächste Mal keinen Treffer zählen.

```
public String prüfDich(String stringTipp) {

    int tipp = Integer.parseInt(stringTipp);

    String ergebnis = "Vorbei";

    for (int zelle : zellorte) {

      if (tipp == zelle) {

        ergebnis = "Treffer";

        anzahlTreffer++;

        break;

      } // if-Ende

    } // for-Ende

    if (anzahlTreffer == zellorte.length) {

      ergebnis = "Versenkt";

    } // if-Ende

    System.out.println(ergebnis);

    return ergebnis;

} // Methodenende
```

Den String in ein int umwandeln.

Eine Variable erstellen, die das Ergebnis festhält, das wir zurückliefern möchten. Als Default-Wert >>Vorbei<< einsetzen (d.h., wir gehen von >>Vorbei<< aus).

Mit jeder Zelle im Array zellorte (jedem Zellort im Objekt) wiederholen.

Dieses Array-Element (Zelle) mit dem Benutzervorschlag vergleichen.

Wenn wir einen Treffer haben!

Aus der Schleife ausbrechen, das Prüfen der anderen Zellen ist überflüssig.

Wir haben die Schleife verlassen und prüfen, ob wir jetzt >>versenkt<< sind (dreimal getroffen wurden), und ändern das String-Ergebnis gegebenenfalls in >>Versenkt<<.

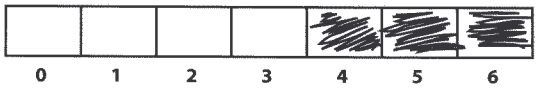
Dem Benutzer das Ergebnis anzeigen (>>Vorbei<<, wenn es nicht in >>Treffer<< oder >>Versenkt<< geändert wurde).

Das Ergebnis an die aufrufende Methode zurückliefern.

# Wie reparieren wir das?

Wir brauchen eine Möglichkeit festzustellen, ob eine Zelle zuvor schon getroffen wurde. Gehen wir mal einige Möglichkeiten durch. Aber erst sehen wir uns an, was wir bisher wissen ...

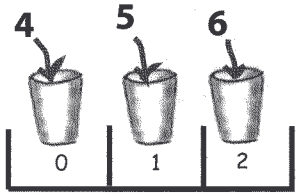
Wir haben eine virtuelle Zeile mit 7 Zellen, in der ein Dot-Com drei beliebige aufeinander folgende Zellen einnimmt. Diese virtuelle Zeile zeigt ein Dot-Com, das sich an den Zellorten 4, 5 und 6 befindet.




Die virtuelle Zeile mit den 3 Zellorten für das DotCom-Objekt.

Das Dot-Com hat eine Instanzvariable – ein int-Array –, die die Zellorte des DotCom-Objekts festhält.

**zellorte**
(Instanzvariable des Dot-Com)



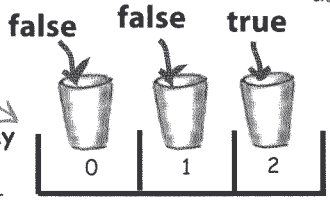

Die Array-Instanzvariable, die die Zellorte des Dot-Com festhält. Dieses Dot-Com enthält die drei Werte 4, 5 und 6. Das sind die Zahlen, die der Benutzer erraten muss.

## ① Option eins

Wir können ein zweites Array erstellen und jeden Treffer, den der Benutzer erzielt, in diesem zweiten Array speichern. Dann könnten wir jedes Mal, wenn ein Treffer erzielt wird, prüfen, ob diese Zelle bereits getroffen wurde.

>true< an einer bestimmten Indexposition bedeutet, dass im ANDEREN Array (zellorte) der Zellort an dieser Indexposition bereits getroffen wurde.

**getroffeneZellen-Array**
(das wäre eine neue boolean-Array-Instanzvariable des Dot-Com)



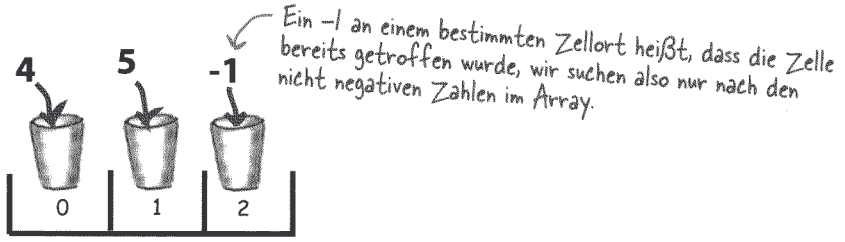

Dieses Array enthält drei Werte, die den Zustand der jeweiligen Zellen im zellorte-Array des Dot-Com repräsentieren. Wird die Zelle bei Index 2 getroffen, würde beispielsweise der Index 2 im Array >>getroffeneZellen<< auf >true< gesetzt.

# Option eins ist zu klobig

Option eins sieht nach mehr Arbeit als erwartet aus. Sie bedeutet, dass Sie jedes Mal, wenn der Benutzer einen Treffer erzielt, den Zustand des zweiten Arrays (des Arrays getroffeneZellen) ändern müssen – ach, und erst müssten Sie natürlich das Array getroffeneZellen PRÜFEN, um nachzusehen, ob diese Zelle nicht schon getroffen wurde. Das würde funktionieren, aber es muss doch eine bessere Lösung geben!

### ② Option zwei

Wir könnten einfach das eine ursprüngliche Array beibehalten, aber darin den Wert der getroffenen Zellen in -1 ändern. Dann haben wir nur noch EIN Array, das wir prüfen und ändern müssen.




**zellorte**
(Instanzvariable von Dot-Com)

# Option zwei ist etwas besser, aber immer noch ziemlich klobig

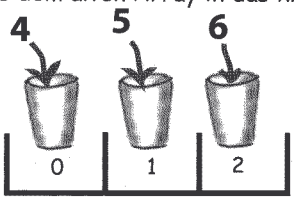
Option zwei ist etwas weniger klobig als Option eins, aber sie ist nicht sonderlich effizient. Sie müssen immer noch alle drei Plätze (Indexpositionen) des Arrays prüfen, obwohl ein oder mehrere Plätze bereits ungültig sind, weil sie schon getroffen wurden (und den Wert -1 haben). Es gibt bestimmt etwas noch Besseres.

Vorcode | Testcode | Hauptcode

## ③ Option drei

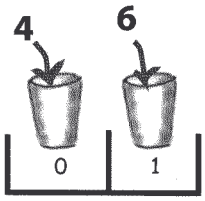
Wir löschen jede Zelle, wenn sie getroffen wird, und verkleinern dann die Größe des Arrays. Das Problem ist nur, dass die Größe von Arrays nicht geändert werden kann. Also müssen wir ein **neues** Array erstellen und die verbleibenden Zellen aus dem alten Array in das kleinere neue Array kopieren.

**zellorte-Array**
BEVOR eine Zelle getroffen wurde




Das Array beginnt mit der Größe 3, und wir durchlaufen alle 3 Zellen (Positionen im Array), um nach einer Übereinstimmung zwischen dem Rateversuch des Benutzers und einem Zellwert (4, 5, 6) zu suchen.

**zellorte-Array**
NACHDEM die Zelle 5, die sich bei Index 1 des Arrays befand, getroffen wurde




Wenn Zelle 5 getroffen wird, erzeugen wir ein neues kleineres Array, das nur noch die verbleibenden Zellorte enthält, und weisen es der ursprünglichen zellorte-Referenz zu.

**Option drei wäre viel besser, wenn man das Array schrumpfen könnte. Dann müssten wir nicht erst ein neues kleineres Array erstellen, die verbleibenden Werte rüberkopieren und die Referenz neu zuweisen.**

### Der ursprüngliche Vorcode-Teil für die Methode prüfDich():

**WIEDERHOLE** mit jedem der Zellorte im *int*-Array.
*// VERGLEICHE* den Rateversuch des Benutzers mit dem Zellort.
**WENN** der Rateversuch des Benutzers passt:
**INKREMENTIERE** die Anzahl von Treffern.
*// ERMITTLE*, ob es der letzte Zellort war:
**WENN** Anzahl der Treffer gleich 3, **LIEFER** »Versenkt« als Ergebnis.
**SONST** nicht versenkt, also **LIEFER** »Treffer«.
ENDE WENN
**SONST** der Rateversuch des Benutzers passt nicht, also **LIEFER** »Vorbei«.
ENDE WENN
ENDE WIEDERHOLE

### Wäre das Leben schön, wenn wir das einfach so ändern könnten:

**WIEDERHOLE** mit jedem *verbleibenden* Zellort.
*// VERGLEICHE* den Rateversuch des Benutzers mit dem Zellort.
**WENN** der Rateversuch des Benutzers passt:
**ENTFERNE** diese Zelle aus dem Array.
*// ERMITTLE*, ob es der letzte Zellort war:
**WENN** das Array jetzt leer ist, **LIEFER** »Versenkt« als Ergebnis.
**SONST** nicht versenkt, also **LIEFER** »Treffer«.
ENDE WENN
**SONST** der Rateversuch des Benutzers passt nicht, also **LIEFER** »Vorbei«.
ENDE WENN
ENDE WIEDERHOLE

Wenn ich nur ein Array finden könnte, das **schrumpfen kann,** wenn man etwas **entfernt**. Und vielleicht auch gleich eins, das man nicht immer vollständig durchlaufen muss, um jedes Element unter die Lupe zu nehmen, eins eben, das man einfach fragen kann, **ob es das enthält,** was man sucht. Und dann sollte man auch noch Dinge **rausholen** können, ohne dass man wissen muss, an welchem Platz das Ding steckt. Das wäre traumhaft. Aber es ist eben nur ein Traum ...

# Wachen Sie auf und schnuppern Sie an der Java-API

**Und auch wenn es wie Zauberei erscheint – so etwas gibt es wirklich.**

**Allerdings ist es kein *Array*, sondern eine *ArrayList*.**

**Eine Klasse in der Java-Kernbibliothek (der API).**

Die Java Standard Edition (die Sie wahrscheinlich verwenden, es sei denn, Sie arbeiten mit der Micro Edition für Kleingeräte – und glauben Sie mir, *das wüssten Sie*) wird mit Hunderten vorgefertigter Klassen ausgeliefert. Diese sind genau wie unsere Code-Fertiggerichte, nur dass diese eingebauten Klassen bereits kompiliert sind.

***Das heißt: keine Tipperei!***

Nutzen Sie sie einfach.

Eine von einer Unmenge an Klassen in der Java-Bibliothek.

Sie können sie in Ihrem Code verwenden, als hätten Sie sie selbst geschrieben.

**ArrayList**

**add(Object elem)**
Fügt der Liste das durch den Parameter angegebene Objekt hinzu.

**remove(int index)**
Entfernt das Objekt bei dem durch den Parameter angegebenen Index.

**remove(Object elem)**
Entfernt dieses Objekt (wenn es sich in der ArrayList befindet).

**contains(Object elem)**
Liefert »true« zurück, wenn es einen Treffer für das Objekt gibt, das durch den Parameter angegeben wird.

**isEmpty()**
Liefert »true« zurück, wenn die Liste keine Elemente enthält.

**indexOf(Object elem)**
Liefert entweder den Index für das durch den Parameter angegebene Objekt oder –1 zurück.

**size()**
Liefert die Anzahl der Elemente zurück, die sich aktuell in der Liste befinden.

**get(int index)**
Liefert das Objekt zurück, das sich aktuell an dem durch den Parameter angegebenen Index befindet.

Das ist nur ein Auszug mit EINIGEN der Methoden in ArrayList.

(Hinweis: Die Methode add(Object elem) sieht eigentlich etwas seltsamer aus, als wir sie hier angegeben haben ... die richtige werden wir uns später im Buch ansehen. Betrachten Sie sie im Augenblick einfach als eine add()-Methode, die das Objekt entgegennimmt, das Sie hinzufügen möchten.)

# Ein paar Dinge, die Sie mit ArrayList machen können

1. Eine erstellen

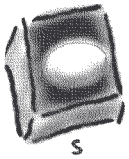
Machen Sie sich jetzt noch keine Gedanken über diese neue <Ei>-Syntax mit den spitzen Klammern. Es heißt einfach »mach die Liste zu einer Liste mit Ei-Objekten«.

```
ArrayList<Ei> meineListe = new ArrayList<Ei>();
```

Ein neues ArrayList-Objekt wird auf dem Heap angelegt. Es ist klein, weil es leer ist.

2. Etwas reinstecken

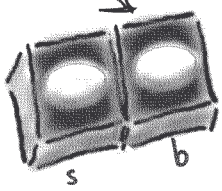
```
Ei s = new Ei();

meineListe.add(s);
```




Jetzt wächst die ArrayList um ein »Kästchen«, das das Ei-Objekt aufnimmt.

3. Etwas anderes reinstecken

```
Ei b = new Ei();

meineListe.add(b);
```




Die ArrayList wächst weiter, um das zweite Ei-Objekt aufzunehmen.

4. Ermitteln, wie viele Dinge darin stecken

```
int dieGröße = meineListe.size();
```

Die ArrayList enthält 2 Objekte, also liefert die Methode size() 2 zurück.

5. Herausfinden, ob sie etwas enthält

```
boolean istDarin = meineListe.contains(s);
```

Die ArrayList ENTHÄLT das Ei-Objekt, das von »s« referenziert wird, also liefert contains() true zurück.

6. Herausfinden, wo sich etwas befindet (d.h. seinen Index)

```
int idx = meineListe.indexOf(b);
```

ArrayList ist nullbasiert (das bedeutet, dass der erste Index 0 ist), und da das Objekt, das von »b« referenziert wird, der Liste als zweites hinzugefügt wurde, liefert indexOf() 1 zurück.

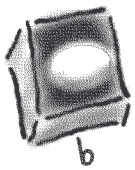
7. Ermitteln, ob sie leer ist

```
boolean leer = meineListe.isEmpty();
```

Die Liste ist definitiv NICHT leer, also liefert isEmpty() false zurück.

8. Etwas daraus entfernen

```
meineListe.remove(s);
```




Schau dir das an – sie ist geschrumpft!

Füllen Sie den Rest der Tabelle aus, indem Sie auf den ArrayList-Code auf der linken Seite schauen und rechts den Code ergänzen, den Sie schreiben müssten, wenn Sie stattdessen ein gewöhnliches Array verwendeten. Wir erwarten nicht, dass Sie das jetzt schon alles richtig machen. Geben Sie einfach Ihr Bestes.

| ArrayList | Gewöhnliches Array |
|---|---|
| `ArrayList<String> meineListe = new ArrayList<String>();` | String [] meineListe = new String[2]; |
| | |
| `String a = new String("huhu");` | String a = new String("huhu"); |
| `meineListe.add(a);` | |
| | |
| `String b = new String("Frosch");` | String b = new String("Frosch"); |
| `meineListe.add(b);` | |
| | |
| `int dieGröße = meineListe.size();` | |
| | |
| `Object o = meineListe.get(1);` | |
| | |
| `meineListe.remove(1);` | |
| | |
| `boolean istDarin = meineListe.contains(b);` | |

## Es gibt keine Dummen Fragen

**F: ArrayList ist schon cool, aber wie erfahre ich, dass es so etwas überhaupt gibt?**

A: Die Frage ist eigentlich: »Wie *erfahre* ich, was in der API ist?«, und das ist der Schlüssel zu Ihrem Erfolg als Java-Programmierer. Ganz zu schweigen davon, dass es der Schlüssel dazu ist, so faul wie möglich sein und trotzdem erfolgreich Software zu kreieren. Sie wären überrascht, wenn Sie wüssten, wie viel Zeit Sie sparen könnten, wenn jemand anderer bereits die meiste Schwerarbeit geleistet hat und Sie nur noch dazukommen und die angenehmen Dinge erledigen müssten. Aber wir schweifen ab ... die kurze Antwort ist, dass Sie einfach etwas Zeit damit verbringen müssen zu lernen, was in der Kern-API steckt. Die lange Antwort ist, dass Sie bis zum Ende des Kapitels lernen werden, *wie* Sie das machen.

**F: Aber das ist eine ziemlich große Sache. Ich muss nicht nur wissen, dass die Java-Bibliothek eine ArrayList bietet. Noch wichtiger ist doch, dass ich weiß, dass ArrayList das Ding ist, das das kann, was ich suche! Wie also komme ich von der Notwendigkeit zum Weg, etwas mit Hilfe der API zu tun?**

A: Jetzt sind Sie wirklich zum Kern der Sache vorgestoßen. Nach dem Durcharbeiten dieses Buchs haben Sie einen guten Überblick über die Sprache. Bei dem, was Ihnen dann noch zu lernen bleibt, geht es eigentlich nur noch darum zu wissen, wie Sie mit dem geringsten Aufwand (mit so wenig Code wie möglich) von einem Problem zu der passenden Lösung kommen. Vielleicht gedulden Sie sich nur noch ein paar Seiten – am Ende dieses Kapitels werden wir darauf eingehen.

## Java im Gespräch

### Interview der Woche: ArrayList über Arrays

**Von Kopf bis Fuß:** ArrayLists sind also wie Arrays, oder?

**ArrayList:** Das hätten die wohl gern! ***Ich*** bin ein *Objekt*, das ist ein himmelweiter Unterschied.

**Von Kopf bis Fuß:** Wenn ich mich nicht irre, sind Arrays auch Objekte. Sie leben doch auf dem Heap wie alle anderen Objekte.

**ArrayList:** Stimmt schon. Arrays kommen auf den Heap, ***bah***, aber ein Array ist trotzdem nur eine Möchtegern-ArrayList. Ein Blender. Objekte haben einen Zustand *und* ein Verhalten, oder? Da sind wir uns ja wohl einig. Aber haben Sie tatsächlich schon einmal versucht, eine Methode auf einem Array aufzurufen?

**Von Kopf bis Fuß:** Wo Sie das jetzt so sagen – eigentlich kann ich nicht behaupten, dass ich das schon mal gemacht habe. Aber welche Methoden sollte ich den überhaupt aufrufen? Ich interessiere mich doch nur für das Zeug, das ich *in* das Array gesteckt habe, nicht für das Array selbst. Und ich kann die Array-Syntax verwenden, wenn ich etwas in ein Array stecken oder etwas aus einem Array herausholen will.

**ArrayList:** Wirklich? Wollen Sie mir etwa sagen, dass Sie tatsächlich schon mal etwas aus einem Array *entfernt* haben? (Meine Güte, wo *lernen* die euch Typen an? Bei McJava?)

**Von Kopf bis Fuß:** *Natürlich* hol ich etwas aus einem Array raus. Ich sag Hund h = hundArray[1] und krieg das Hund-Objekt am Index 1 des Arrays.

**ArrayList:** Na vielleicht erklär ich das mal ganz langsam. Vielleicht können Sie mir dann folgen. Sie haben diesen Hund *nicht*, ich wiederhole, *nicht* aus dem Array entfernt. Sie haben bloß eine Kopie der *Referenz auf den Hund* kopiert und weisen sie einer anderen Hund-Variablen zu.

**Von Kopf bis Fuß:** Ah, jetzt verstehe ich, was Sie meinen. Nein, ich habe das Hund-Objekt nicht wirklich aus dem Array entfernt. Es ist immer noch da. Aber ich nehme an, ich könnte einfach seine Referenz auf null setzen.

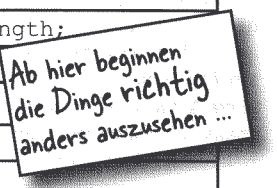
**ArrayList:** Aber ich bin ein erstrangiges Objekt und habe Methoden und kann tatsächlich Dinge *tun*, wie die Hund-Referenz aus mir zu entfernen, anstatt sie einfach auf null zu setzen. Und ich kann meine Größe *dynamisch* (das schlagen Sie vielleicht besser nach) ändern. Versuchen Sie mal, ein *Array* dazu zu bringen!

**Von Kopf bis Fuß:** Äh, ich frag das wirklich ungern – aber es gibt das Gerücht, dass Sie nichts anderes sind als ein besseres, aber weniger effizientes Array. Dass Sie eigentlich nur eine Art Verpackung – ein »Wrapper« – für ein Array sind, die ein paar zusätzliche Methoden wie die Änderung der Größe hinzufügt, damit ich die nicht selbst schreiben muss. Und wo wir gerade dabei sind, *Sie können nicht mal elementare Typen speichern*! Ist das nicht eine große Beschränkung?

**ArrayList:** Ich kann nicht *glauben*, dass Sie diesem Stammtischgebrabbel Glauben schenken. Nein, ich bin *nicht* bloß ein weniger effizientes Array. Ich gebe zu, dass es ein paar *sehr, sehr seltene* Situationen gibt, in denen ein Array bei bestimmten Dingen eine Spur, ich sage es noch mal, *eine Spur* schneller sein kann. Aber ist es dieser *winzige* Leistungsvorsprung wert, die ganze *Macht* aufzugeben? Sehen Sie sich doch mal all diese *Flexibilität* an. Und das mit den elementaren Typen – *natürlich* können Sie elementare Typen in eine ArrayList packen, Sie müssen sie einfach nur in eine der Wrapper-Klassen für elementare Typen packen (darüber werden Sie in Kapitel 10 noch eine Menge erfahren). Und seit Java 5.0 erfolgt dieses Wrapping automatisch (wie auch das Unwrapping, wenn Sie einen elementaren Typen wieder herausnehmen). Und, stimmt schon, ich *geb ja zu*, dass es stimmt, dass ein Array mit elementaren Typen wahrscheinlich schneller wäre als eine ArrayList mit *elementaren Typen* wegen des Wrappings und Unwrapping, aber ... wer verwendet denn *heutzutage* noch diese elementaren Typen? Oh, wie spät es schon ist! *Ich komm zu spät zu Pilates*. Wir müssen das wohl ein anderes Mal fortsetzen.

# Ein Vergleich zwischen ArrayList und einem gewöhnlichen Array

| ArrayList | Gewöhnliches Array |
|---|---|
| `ArrayList<String> meineListe = new ArrayList<String>();` | `String [] meineListe = new String[2];` |
| | |
| `String a = new String("huhu");` | `String a = new String("huhu");` |
| `meineListe.add(a);` | `meineListe[0] = a;` |
| | |
| `String b = new String("Frosch");` | `String b = new String("Frosch");` |
| `meineListe.add(b);` | `meineListe[1] = b;` |
| | |
| `int dieGröße = meineListe.size();` | `int dieGröße = meineListe.length;` |
| | |
| `Object o = meineListe.get(1);` | `String o = meineListe[1];` |
| | |
| `meineListe.remove(1);` | `meineListe[1] = null;` |
| | |
| `boolean istDarin = meineListe.contains(b);` | `boolean istDarin = false;`<br>`for (String element : meineListe) {`<br>`if (b.equals(element)) {`<br>`istDarin = true;`<br>`break;`<br>`}`<br>`}` |




Sie sehen, dass Sie bei ArrayList mit einem Objekt vom Typ ArrayList arbeiten. Sie rufen also einfach nur mit dem guten alten Punktoperator die guten alten Methoden auf einem guten alten Objekt auf.

Bei einem *Array* verwenden Sie die *spezielle Array-Syntax* (wie meineListe[0] = foo), die Sie nirgendwo sonst als bei Arrays verwenden. Obwohl ein Array ein Objekt *ist*, lebt es in seiner eigenen besonderen Welt. Arrays besitzen keine eigenen speziellen Methoden, die Sie zur Arbeit mit einem Array aufrufen könnten, sondern nur eine einzige Instanzvariable, *length*.

# Ein Vergleich zwischen ArrayList und einem gewöhnlichen Array

### 1 Ein simples altes Array muss seine Größe bereits kennen, wenn es erzeugt wird.

Bei einer ArrayList erzeugen Sie dagegen einfach ein Objekt vom Typ ArrayList. Jedes Mal. Sie muss nie wissen, wie groß sie sein soll, weil sie wächst und schrumpft, wenn Objekte hinzugefügt oder entfernt werden.

```
new String[2]
```

Braucht eine Größe.

```
new ArrayList<String>()
```

Keine Größe erforderlich (obwohl Sie eine Größe angeben können, wenn Sie das möchten).

### 2 Wenn Sie ein Objekt in ein gewöhnliches Array stecken möchten, müssen Sie es einem bestimmten Ort zuweisen.

(Einem Index zwischen 0 und der Länge des Arrays minus eins.)

```
meineListe[1] = b;
```

Braucht einen Index.

Wenn dieser Index außerhalb der Grenzen des Arrays liegt (beispielsweise wenn Sie bei einem Array, das laut Deklaration die Größe 2 haben soll, versuchen, dem Index 3 etwas zuzuweisen), geht es zur Laufzeit hoch.

Bei ArrayList können Sie einen Index angeben, indem Sie die Methode *add(einInt, einObjekt)* verwenden. Aber Sie können auch weiterhin einfach *add(einObjekt)* sagen, und die ArrayList wächst einfach weiter, um Platz für das neue Ding zu schaffen.

```
meineListe.add(b);
```

Kein Index.

### 3 Arrays verwenden die Array-Syntax, die in Java an keiner anderen Stelle verwendet wird.

Aber ArrayLists sind ganz gewöhnliche Java-Objekte. Es gibt also keine besondere Syntax.

```
meineListe[1]
```

Die eckigen Array-Klammern [ ] sind eine besondere Syntax nur für Arrays.

### 4 ArrayLists in Java 5.0 sind parametrisiert.

Wir haben gerade gesagt, dass ArraysLists im Unterschied zu Arrays keine besondere Syntax haben. Aber sie *verwenden* etwas Besonderes, das in Java 5.0 Tiger hinzugefügt wurde – ***parametrisierte Typen.***

```
ArrayList<String>
```

Dieser <String> in den spitzen Klammern ist ein »Typparameter«. ArrayList<String> bedeutet einfach »eine Liste mit Strings«, während ArrayList<Hund> »eine Liste mit Hunden« meint.

Vor Java 5.0 gab es keine Möglichkeit, den *Typ* der Dinge zu deklarieren, die in eine ArrayList kommen sollten. Für den Compiler waren also alle ArrayLists einfach heterogene Sammlungen von Objekten. Aber jetzt können wir mit der <HierKommtDerTyp-Hin>-Syntax eine ArrayList deklarieren und erzeugen, die weiß (und beschränkt), welche Typen von Objekten sie aufnehmen kann. Einzelheiten zu parametrisierten Typen in ArrayLists werden wir uns im Collections-Kapitel ansehen. Im Augenblick sollten Sie über die Spitze-Klammern-Syntax <>, die wir verwenden, wenn wir ArrayLists verwenden, nicht zu viel nachdenken. Merken Sie sich einfach, dass sie ein Mittel ist, den Compiler dazu zu zwingen, in der ArrayList nur einen bestimmten Typ von Objekt zu erlauben *(den Typ in den spitzen Klammern).*

# Reparieren wir den DotCom-Code.

**Erinnern Sie sich? So sah die fehlerhafte Version aus:**

Wir haben die Klasse jetzt in DotCom (statt EinfachesDotCom) umbenannt, weil wir uns nun an die verbesserte Version machen, aber der Code ist der gleiche wie der, den Sie im vorangegangenen Kapitel sahen.

```
public class DotCom {

   int[] zellorte;
   int anzahlTreffer;

   public void setZellorte(int[] orte) {
      zellorte = orte;
   }

   public String prüfDich(String stringTipp) {
      int tipp = Integer.parseInt(stringTipp);
      String ergebnis = "Vorbei";

      for (int zelle : zellorte) {
         if (tipp == zelle) {
            ergebnis = "Treffer";
            anzahlTreffer++;
            break;
         }
      } // for-Ende

      if (anzahlTreffer == zellorte.length) {
         ergebnis = "Versenkt";
      } // if-Ende
      System.out.println(ergebnis);
      return ergebnis;
   } // Methodenende
} // Klassenende
```

Hier ging die Sache schief. Wir haben jeden Rateversuch als Treffer gezählt, ohne zu prüfen, ob die Zelle bereits getroffen wurde.

Vorcode | Testcode | Hauptcode

# Die neue und verbesserte DotCom-Klasse

```
import java.util.ArrayList;

public class DotCom {

   private ArrayList<String> zellorte;
   // int anzahlTreffer;
   // brauchen wir jetzt nicht mehr

   public void setZellorte(ArrayList<String> orte) {
      zellorte = orte;
   }

   public String prüfDich(String benutzerEingabe) {

      String ergebnis = "Vorbei";

      int index = zellorte.indexOf(benutzerEingabe);

      if (index >= 0) {

         zellorte.remove(index);


         if (zellorte.isEmpty()) {
           ergebnis = "Versenkt";
         } else {
           ergebnis = "Treffer" ;
         } // inneres if schließen

       } // äußeres if schließen
       System.out.println(ergebnis);

       return ergebnis;
   } // Methodenende
} // Klassenende
```

Ignorieren Sie diese Zeile jetzt erst einmal. Darüber werden wir am Ende des Kapitels reden.

Das Array in eine ArrayList ändern, die Strings enthält.

Neuer, besserer Parametername.

Ermitteln, ob der Rateversuch des Benutzers sich in der ArrayList befindet, indem nach dem Index dafür gefragt wird. Ist er nicht in der Liste, liefert indexOf() den Wert -1 zurück.

Wenn index größer oder gleich 0 ist, befindet sich der Rateversuch des Benutzers definitiv in der Liste, also wird das entsprechende Element aus der Liste entfernt.

Wenn die Liste leer ist, war das der Schlag, der das Dot-Com versenkt hat!

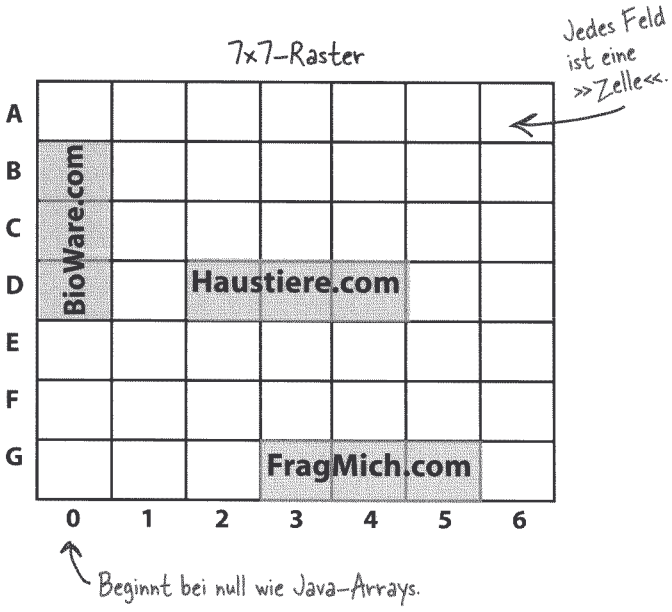
# Bauen wir das RICHTIGE Spiel auf: »Dot-Com versenken«

Bisher haben wir an der ›vereinfachten‹ Version gearbeitet, aber jetzt wollen wir die richtige schreiben. An Stelle einer einzigen Zeile werden wir jetzt ein Raster verwenden. Und statt eines Dot-Com werden wir drei verwenden.

**Ziel**: Alle Dot-Coms des Computers mit so wenigen Versuchen wie möglich versenken. Sie erhalten eine Einstufung auf Basis dessen, wie Sie sich anstellen.

**Setup**: Wenn das Spielprogramm gestartet wird, platziert der Computer drei Dot-Coms zufällig auf dem **virtuellen 7x7-Raster**. Ist dieser Vorgang abgeschlossen, fordert das Spiel Sie zu einem ersten Rateversuch auf.

**Wie Sie spielen:** Wir haben noch nicht gelernt, wie man eine GUI aufbaut, also arbeitet diese Version auf der Kommandozeile. Der Computer fordert Sie auf, Ihren Rateversuch (eine Zelle) einzugeben, den Sie auf der Kommandozeile (»A3«, »C5« usw.) eintippen. Als Antwort auf Ihren Rateversuch sehen Sie auf der Kommandozeile ein Ergebnis, entweder »Treffer«, »Vorbei« oder »Sie haben Haustiere.com versenkt« (oder wie das Dot-Com der Stunde auch immer heißen mag). Wenn Sie alle drei Dot-Coms in den großen 404-Himmel geblasen haben, endet das Spiel damit, dass Ihnen eine Bewertung ausgegeben wird.




**Sie schreiben das Spiel Dot-Com versenken mit einem 7x7-Raster und drei Dot-Coms. Jedes Dot-Com nimmt drei Zellen ein.**

## Teil eines Spielablaufs

```
Datei Bearbeiten Fenster Hilfe Verkaufen
%java DotComVersenken
Geben Sie einen Rateversuch ein  D4
Treffer
Geben Sie einen Rateversuch ein  D5
Vorbei
Geben Sie einen Rateversuch ein  E4
Vorbei
Geben Sie einen Rateversuch ein  D3
Treffer
Geben Sie einen Rateversuch ein  D2
Grrr! Sie haben Haustiere.com versenkt : (
Versenkt
Geben Sie einen Rateversuch ein  B2
Vorbei
Geben Sie einen Rateversuch ein  G1
Vorbei
Geben Sie einen Rateversuch ein  G4
Treffer
Geben Sie einen Rateversuch ein  F4
Vorbei
Geben Sie einen Rateversuch ein  G3
Treffer
Geben Sie einen Rateversuch ein  G5
Grrr! Sie haben FragMich.com versenkt : (
Versenkt
Geben Sie einen Rateversuch ein  B6
Vorbei
Geben Sie einen Rateversuch ein  G0
```

# Was muss geändert werden?

Wir haben drei Klassen, die geändert werden müssen: die DotCom-Klasse (die jetzt DotCom statt Einfaches-DotCom heißt), die Spielklasse (DotComVersenken) und die Spiel-Helferklasse (über die wir uns jetzt noch keine Gedanken machen).

## A DotCom-Klasse

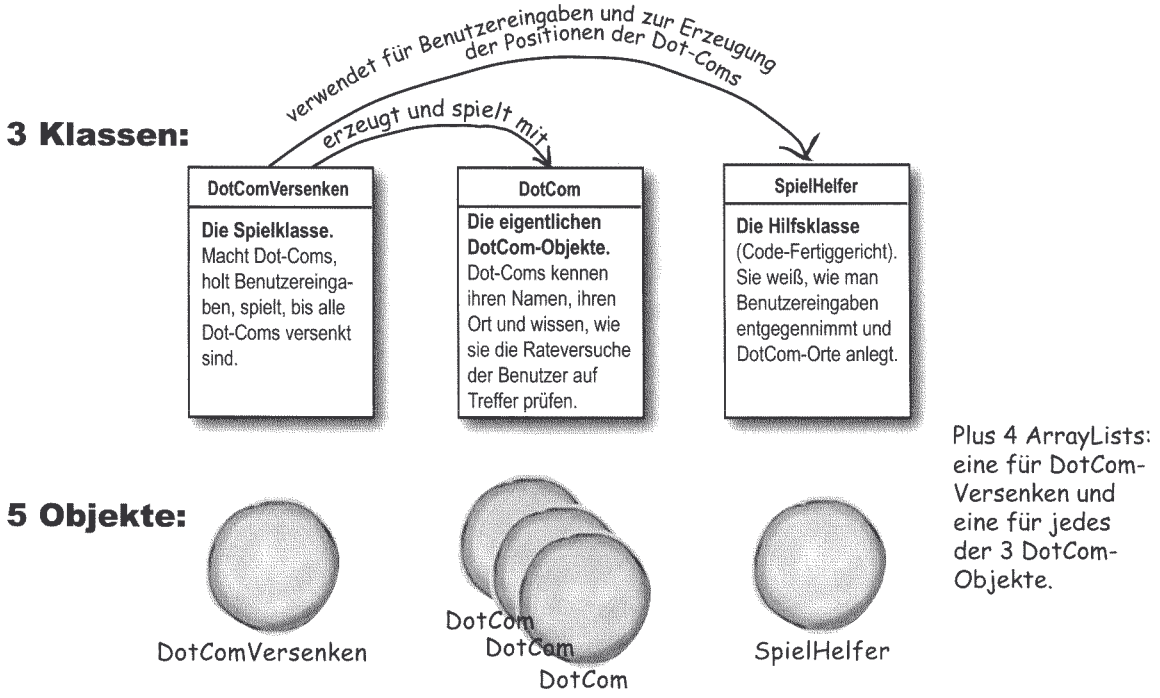
- **Eine Variable *name* hinzufügen ...**
  ... die den Namen des Dot-Com festhalten soll (»Haustiere.com«, »BioWare.com« usw.), damit jedes Dot-Com seinen Namen ausgeben kann, wenn es versenkt wird (wie Sie es in der Ausgabe auf der vorangegangenen Seite sehen).

## B DotComVersenken-Klasse (das Spiel)

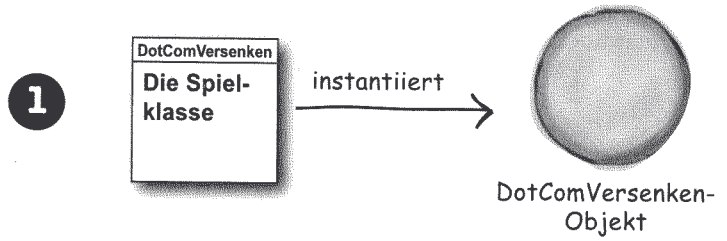
- ***Drei* Dot-Coms erzeugen an Stelle von *einem*.**
- **Jedem der drei Dot-Coms einen *Namen* geben.**
  Auf jeder DotCom-Instanz eine Setter-Methode aufrufen, damit das Dot-Com seiner name-Instanzvariablen den Namen zuweisen kann.

## DotComVersenken-Klasse (Forts.)

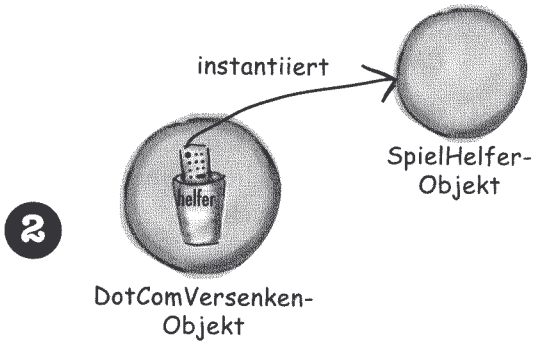
- **Die drei Dot-Coms in einem Raster statt auf einer einzelnen Zeile platzieren.**
  Dieser Schritt ist jetzt weit komplexer als vorher, weil wir die Dot-Coms zufällig positionieren wollen. Da wir uns hier nicht mit den Berechnungen herumschlagen wollen, packen wir den Algorithmus, mit dem wir den Dot-Coms einen Ort zuweisen, in die Klasse SpielHelfer (ein Fertiggericht).
- **Alle Rateversuche des Benutzers *an allen drei Dot-Coms* prüfen statt nur an einem.**
- **Das Spiel weiterspielen** (d.h. weitere Rateversuche entgegennehmen und an den verbleibenden Dot-Coms prüfen), ***bis kein Dot-Com mehr über Wasser ist.***
- **main() hinter uns lassen.** Wir haben die einfache Version nur in main() gehalten ... um sie einfach zu halten. Aber für das *richtige* Spiel wollen wir das nicht.



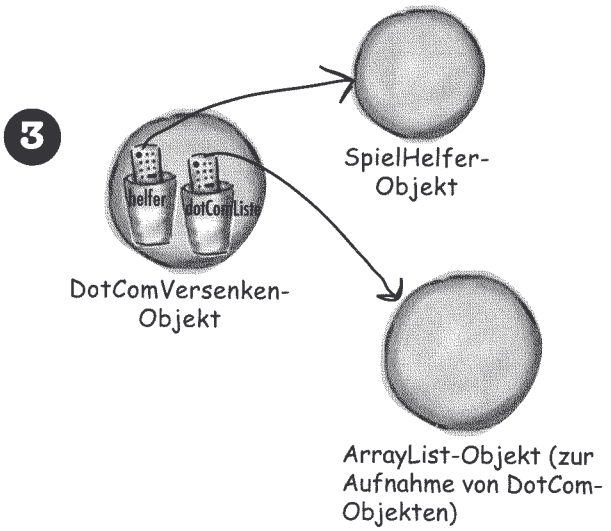
# Wer macht was im Spiel DotComVersenken
## (und wann)



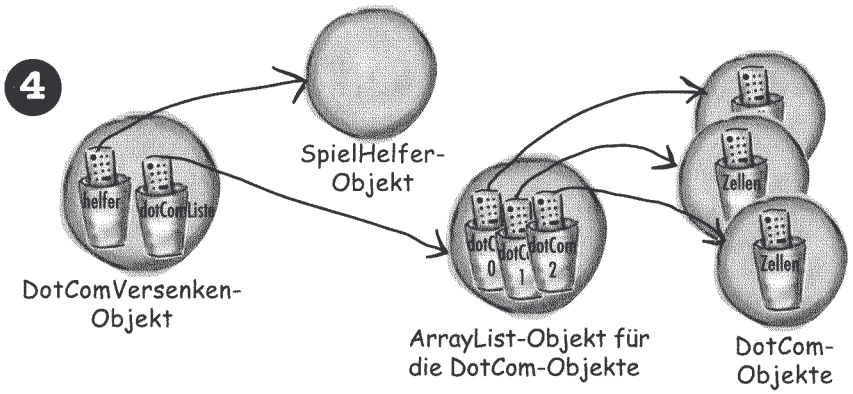

Die main()-Methode in der Klasse DotComVersenken instantiiert das DotCom-Versenken-Objekt, das den ganzen Spielkram erledigt.




Das DotComVersenken-Objekt (die Spielklasse) instantiiert eine Instanz von SpielHelfer, das Objekt, das das Spiel dabei unterstützt, seine Arbeit zu erledigen.



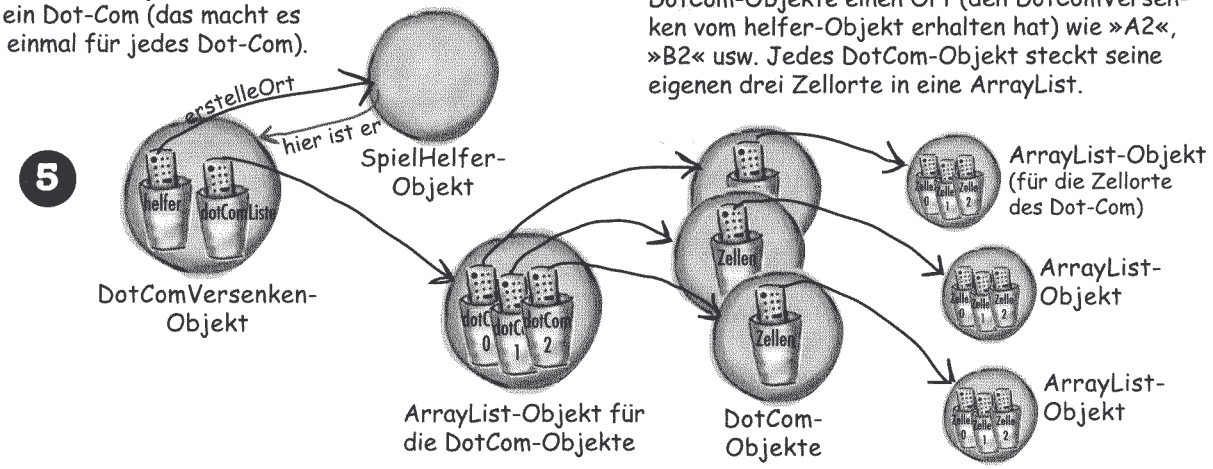

Das DotComVersenken-Objekt instantiiert eine ArrayList, die die 3 DotCom-Objekte festhält.




Das DotComVersenken-Objekt erzeugt drei Dot-Com-Objekte (und steckt sie in die ArrayList).

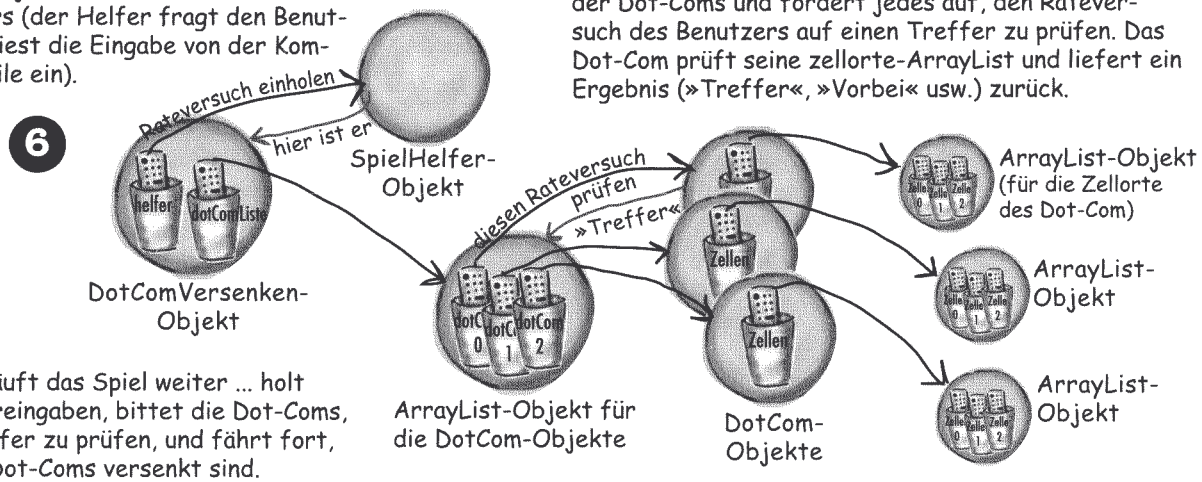
Das DotComVersenken-Objekt bittet das helfer-Objekt um einen Ort für ein Dot-Com (das macht es dreimal, einmal für jedes Dot-Com).

Das DotComVersenken-Objekt gibt jedem der DotCom-Objekte einen Ort (den DotComVersenken vom helfer-Objekt erhalten hat) wie »A2«, »B2« usw. Jedes DotCom-Objekt steckt seine eigenen drei Zellorte in eine ArrayList.

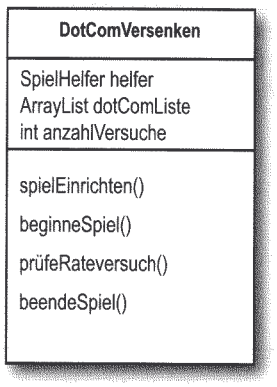



Das DotComVersenken-Objekt bittet das helfer-Objekt um einen Rateversuch des Benutzers (der Helfer fragt den Benutzer und liest die Eingabe von der Kommandozeile ein).

Das DotComVersenken-Objekt durchläuft die Liste der Dot-Coms und fordert jedes auf, den Rateversuch des Benutzers auf einen Treffer zu prüfen. Das Dot-Com prüft seine zellorte-ArrayList und liefert ein Ergebnis (»Treffer«, »Vorbei« usw.) zurück.




Und so läuft das Spiel weiter ... holt Benutzereingaben, bittet die Dot-Coms, auf Treffer zu prüfen, und fährt fort, bis alle Dot-Coms versenkt sind.




# Vorcode für die echte DotComVersenken-Klasse

Die Klasse DotComVersenken hat drei Hauptaufgaben: das Spiel einrichten, das Spiel spielen, bis alle Dot-Coms versenkt sind, und das Spiel beenden. Wir können diese drei Aufgaben zwar auch unmittelbar auf drei Methoden abbilden, ziehen es aber vor, die mittlere Aufgabe (das Spiel spielen) auf *zwei* Methoden zu verteilen, um die Granularität feiner zu halten. Kleinere Methoden (d.h. kleinere Bündel von Funktionalitäten) vereinfachen das Testen, Debuggen und Verändern des Codes.

## Variablen-Deklarationen

**DEKLARATION** (und Instantiierung)**:** ***SpielHelfer***-Instanzvariable namens ***helfer.***

**DEKLARATION** (und Instantiierung)**:** ***ArrayList*** namens ***dotComListe*** für die Liste mit den Dot-Coms (anfänglich drei).

**DEKLARATION:** int-Variable für die Anzahl der Rateversuche (damit wir dem Benutzer bei Spielende eine Bewertung geben können). Nenne sie ***anzahlVersuche*** und setze sie auf 0.

## Methoden-Deklarationen

**DEKLARATION:** ***spielEinrichten()***-Methode, um die DotCom-Objekte zu erzeugen und mit einem Namen und einem Ort zu initialisieren. Dem Benutzer eine kurze Spielanleitung anzeigen.

**DEKLARATION:** ***beginneSpiel()***-Methode, die den Benutzer zur Eingabe eines Rateversuchs auffordert und die prüfeRateversuch()-Methode aufruft, bis alle DotCom-Objekte aus dem Spiel entfernt sind.

**DEKLARATION:** ***prüfeRateversuch()***-Methode, die alle verbleibenden DotCom-Objekte durchläuft und ihre prüfDich()-Methode aufruft.

**DEKLARATION:** ***beendeSpiel()***-Methode, die eine Meldung mit der Bewertung des Benutzers ausgibt, die darauf basiert, wie viele Versuche er brauchte, um alle DotCom-Objekte zu versenken.

## Methoden-Implementierung

**METHODE:** *void spielEinrichten()*

*// drei DotCom-Objekte erzeugen und ihnen Namen geben*

**ERZEUGE** drei DotCom-Objekte.

**SETZE** einen Namen für jedes Dot-Com.

**FÜGE** die Dot-Coms zur ***dotComListe*** (der ArrayList) hinzu.

**WIEDERHOLE** mit jedem der DotCom-Objekte in der ArrayList ***dotComListe.***

**RUFE** auf dem Helferobjekt die Methode ***platziereDotCom()*** auf, um einen zufällig ausgewählten Ort für dieses Dot-Com zu erhalten (drei vertikal oder horizontal ausgerichtete Zellen in einem 7x7-Raster).

**SETZE** den Ort für jedes Dot-Com auf Grundlage des Ergebnisses des ***platziereDotCom()***-Aufrufs.

ENDE WIEDERHOLE

ENDE METHODE

Vorcode | Testcode | Hauptcode

## Methodenimplementierungen (Fortsetzung):

**METHODE**: ***void beginneSpiel()***

**WIEDERHOLE,** solange Dot-Coms vorhanden sind.

**HOLE** Benutzereingabe durch Aufruf der ***holeBenutzereingabe()***-Methode des Helfers.

**EVALUIERE** den Rateversuch des Benutzers mit der Methode ***prüfeRateversuch()***.

ENDE WIEDERHOLE

ENDE METHODE

**METHODE**: ***void prüfeRateversuch(String rateversuch)***

*// ermitteln, ob es einen Treffer (und ein Versenkt) bei einem der Dot-Coms gibt*

**INKREMENTIERE** die Anzahl der Rateversuche des Benutzers in der Variablen ***anzahlVersuche***.

**SETZE** die lokale Variable ***ergebnis*** (einen ***String***) auf »Vorbei« (geht davon aus, dass der Rateversuch »Vorbei« ist).

**WIEDERHOLE** mit jedem der DotCom-Objekte in ***dotComListe***.

**EVALUIERE** den Rateversuch, indem die ***prüfDich()***-Methode des aktuellen DotCom-Objekts aufgerufen wird.

**SETZE** die Variable ergebnis gegebenenfalls auf »Treffer« oder »Versenkt«.

**WENN** ergebnis ist »Versenkt«, **ENTFERNE** Dot-Com aus ***dotComListe***.

ENDE WIEDERHOLE

**ZEIGE** dem Benutzer den Wert von ***ergebnis*** an.

ENDE METHODE

**METHODE**: ***void beendeSpiel()***

**ZEIGE** eine allgemeine »Game Over«-Nachricht an, dann:

**WENN** die Anzahl der Rateversuche klein ist,

**ZEIGE** Glückwunsch-Nachricht an,

**SONST**

**ZEIGE** Verhöhnende Nachricht an.

ENDE WENN

ENDE METHODE

### Spitzen Sie Ihren Bleistift

Wie sollten wir von diesem Vorcode zum endgültigen Code weitergehen? Zuerst beginnen wir mit dem Testcode und dann testen und bauen wir unsere Methoden Steinchen auf Steinchen. Aber wir werden Ihnen in diesem Buch keinen Testcode mehr zeigen. Es ist also Ihre Aufgabe, darüber nachzudenken, was Sie wissen müssen, um diese Methoden zu testen. Und welche Methoden würden Sie als Erstes testen und schreiben? Schauen Sie, ob Sie etwas Vorcode für einen Satz von Tests ausarbeiten können. Vorcode oder eine einfache Aufzählung sind für diese Übung vollkommen ausreichend. Aber wenn Sie versuchen möchten, den *echten* Testcode (in Java) zu schreiben …

## Der DotComVersenken-Code (das Spiel)

Vorcode | Testcode | Hauptcode

```
import java.util.*;
public class DotComVersenken {

  (1) private SpielHelfer helfer = new SpielHelfer();
      private ArrayList<DotCom> dotComListe = new ArrayList<DotCom>();
      private int anzahlVersuche = 0;

  private void spielEinrichten() {
    // erst ein paar Dot-Coms herstellen und ihnen Namen geben
    DotCom eins = new DotCom();
    eins.setName("Haustiere.com");
    DotCom zwei = new DotCom();
    zwei.setName("BioWare.com");
    DotCom drei = new DotCom();
    drei.setName("FragMich.com");
    dotComListe.add(eins);
    dotComListe.add(zwei);
    dotComListe.add(drei);

    System.out.println("Ihre Aufgabe ist es, drei Dot-Coms zu versenken.");
    System.out.println("Haustiere.com, BioWare.com, FragMich.com");
    System.out.println("Versuchen Sie, sie mit so wenig Versuchen wie möglich zu erledigen.");

    for (DotCom aktuellesDotCom : dotComListe) {
        ArrayList<String> neuerOrt = helfer.platziereDotCom(3);
        aktuellesDotCom.setZellorte(neuerOrt);
    } // for-Schleife schließen
  } // Methode spielEinrichten schließen

  private void beginneSpiel(){
    while(!dotComListe.isEmpty()) {
      String rateversuch = helfer.getBenutzereingabe("Geben Sie einen Rateversuch ein");
      prüfeRateversuch(rateversuch)
    } // while schließen
    beendeSpiel();
  } // Methode beginneSpiel schließen
```

(2) (3) (4) (5) (6) (7) (8) (9) (10)

### Spitzen Sie Ihren Bleistift

**Versehen Sie den Code selbst mit Anmerkungen!**

**Ordnen Sie die Anmerkungen am unteren Ende der Seiten den Nummern im Code zu. Schreiben Sie die Nummern vor die passenden Anmerkungen.**

**Jede Anmerkung wird nur einmal verwendet, und Sie brauchen alle Anmerkungen.**

---

Die Variablen deklarieren und initialisieren, die wir brauchen.

Benutzereingabe holen.

Den Helfer nach einem DotCom-Ort (einer ArrayList mit Strings) fragen.

Auf diesem Dot-Com die Setter-Methode aufrufen, um ihm den Ort zu geben, den wir gerade vom Helfer erhalten haben.

Mit allen Dot-Coms in der Liste wiederholen.

Eine kurze Spielanleitung für den Benutzer ausgeben.

Unsere prüfeRateversuch-Methode aufrufen.

Unsere beendeSpiel-Methode aufrufen.

Drei DotCom-Objekte erzeugen, ihnen Namen geben und sie in die ArrayList stecken.

Solange die DotCom-Liste NICHT leer ist (das ! bedeutet NICHT und ist das Gleiche wie dotComListe.isEmpty() == false).

Vorcode | Testcode | Hauptcode

**Blättern Sie auf KEINEN Fall um!**

**Nicht bevor Sie mit dieser Übung fertig sind.**

**Unsere Version finden Sie auf der nächsten Seite.**

```
   private void prüfeRateversuch(String rateversuch) {

      anzahlVersuche++; (11)
      String ergebnis  = "Vorbei"; (12)

      for (DotCom aktuellesDotCom : dotComListe) { (13)
         ergebnis = aktuellesDotCom.prüfDich(rateversuch); (14)
         if (ergebnis.equals("Treffer")) {
               break; (15)
         }
         if (ergebnis.equals("Versenkt")) {
               dotComListe.remove(aktuellesDotCom); (16)
               break;
         }
       } // for schließen
       System.out.println(ergebnis); (17)
   } // Methode schließen

   private void beendeSpiel() {
     System.out.println("Alle Dot-Coms sind versenkt! Ihre Aktien sind wertlos.");
     if (anzahlVersuche <= 18) {
        System.out.println("Sie haben nur " + anzahlVersuche +
                           " Versuche benötigt. Sie erhalten den KinoWelt-Preis!");
     } else {
       System.out.println("Hat ja lang genug gedauert. "+ anzahlVersuche + " Versuche.");
       System.out.println("Warum sind Sie nicht ausgestiegen, bevor der Kurs in den
                           Keller fiel?");
       }
   } // Methode schließen (18)

   public static void main(String[] args) {
     DotComVersenken spiel = new DotComVersenken(); (19)
     spiel.spielEinrichten(); (20)
     spiel.beginneSpiel(); (21)
   } // Methode schließen
}
```

— Mit allen Dot-Coms in der Liste wiederholen.

— Der hier ist versenkt, also nehmen wir ihn aus der DotCom-Liste und steigen dann aus der Schleife aus.

— Eine Nachricht ausgeben, die dem Benutzer sagt, wie gut er sich im Spiel angestellt hat.

— Dem Benutzer das Ergebnis ausgeben.

— Dem Spielobjekt sagen, dass es das Spiel einrichten soll.

— Die Anzahl der Rateversuche des Benutzers inkrementieren.

— Von »Vorbei« ausgehen, bis etwas anderes festgestellt wird.

— Das Dot-Com auffordern, den Rateversuch zu prüfen, um zu sehen, ob es einen Treffer (oder ein Versenkt) gab.

— Die Schleife vorzeitig verlassen, weil es sinnlos ist, die anderen zu prüfen.

— Dem Spielobjekt sagen, dass es die Haupt-Spielschleife starten soll (die die Benutzereingaben abfragt und die Rateversuche prüft).

— Das Spielobjekt erstellen.

## Der DotComVersenken-Code (das Spiel)

Vorcode | Testcode | Hauptcode

```
import java.util.*;
public class DotComVersenken {

    private SpielHelfer helfer = new SpielHelfer();
    private ArrayList<DotCom> dotComListe = new ArrayList<DotCom>();
    private int anzahlVersuche = 0;

    private void spielEinrichten() {
      // erst ein paar Dot-Coms machen und ihnen Namen geben
      DotCom eins = new DotCom();
      eins.setName("Haustiere.com");
      DotCom zwei = new DotCom();
      zwei.setName("BioWare.com");
      DotCom drei = new DotCom();
      drei.setName("FragMich.com");
      dotComListe.add(eins);
      dotComListe.add(zwei);
      dotComListe.add(drei);

      System.out.println("Ihre Aufgabe ist es, drei Dot-Coms zu versenken.");
      System.out.println("Haustiere.com, BioWare.com, FragMich.com");
      System.out.println("Versuchen Sie, sie mit so wenig Versuchen wie möglich zu erledigen.");

      for (DotCom aktuellesDotCom : dotComListe) {

          ArrayList<String> neuerOrt = helfer.platziereDotCom(3);

          aktuellesDotCom.setZellorte(neuerOrt);

      } // for-Schleife schließen
    } // Methode spielEinrichten schließen

    private void beginneSpiel(){

      while(!dotComListe.isEmpty()) {

        String rateversuch = helfer.getBenutzereingabe("Geben Sie einen Rateversuch ein.");
        prüfeRateversuch(rateversuch);

      } // while schließen
      beendeSpiel();
    } // Methode beginneSpiel schließen
```

Die Variablen deklarieren und initialisieren, die wir brauchen.

Eine ArrayList für DotCom-Objekte erzeugen (anders gesagt: eine Liste, die NUR DotCom-Objekte aufnimmt, wie DotCom[] ein Array von DotCom-Objekten angeben würde).

Drei DotCom-Objekte erzeugen, ihnen Namen geben und sie in die ArrayList stecken.

Eine kurze Spielanleitung für den Benutzer ausgeben.

Mit allen Dot-Coms in der Liste wiederholen.

Den Helfer nach einem DotCom-Ort (einer ArrayList mit Strings) fragen.

Auf diesem Dot-Com die Setter-Methode aufrufen, um ihm den Ort zu geben, den wir gerade vom Helfer erhalten haben.

Solange die DotCom-Liste NICHT leer ist (das ! bedeutet NICHT und ist das Gleiche wie dotComListe.isEmpty() == false).

Benutzereingabe holen.

Unsere prüfeRateversuch-Methode aufrufen.

Unsere beendeSpiel-Methode aufrufen.

Vorcode | Testcode | Hauptcode

```
  private void prüfeRateversuch(String rateversuch) {
     anzahlVersuche++;
     String ergebnis  = "Vorbei";
     for (DotCom aktuellesDotCom : dotComListe) {
        ergebnis = aktuellesDotCom.prüfDich(rateversuch);
        if (ergebnis.equals("Treffer")) {
              break;
        }
        if (ergebnis.equals("Versenkt")) {
              dotComListe.remove(aktuellesDotCom);
              break;
        }
     } // for schließen
     System.out.println(ergebnis);
  } // Methode schließen

  private void beendeSpiel() {
    System.out.println("Alle Dot-Coms sind versenkt! Ihre Aktien sind wertlos.");
    if (anzahlVersuche <= 18) {
       System.out.println("Sie haben nur " + anzahlVersuche + " Versuche benötigt.
                          Sie erhalten den KinoWelt-Preis!");
    } else {
       System.out.println("Hat ja lang genug gedauert. "+ anzahlVersuche + " Versuche.");
       System.out.println("Warum sind Sie nicht ausgestiegen, bevor der Kurs in den
                          Keller fiel?");
    }
  }// Methode schließen

  public static void main(String[] args) {
     DotComVersenken spiel = new DotComVersenken();
     spiel.spielEinrichten();
     spiel.beginneSpiel();
  }// Methode schließen
}
```

Die Anzahl der Rateversuche des Benutzers inkrementieren.

Von »Vorbei« ausgehen, bis etwas anderes festgestellt wird.

Mit allen Dot-Coms in der Liste wiederholen.

Das Dot-Com auffordern, den Rateversuch zu prüfen, um zu sehen, ob es einen Treffer (oder ein Versenkt) gab.

Die Schleife vorzeitig verlassen, weil es sinnlos ist, die anderen zu prüfen.

Der hier ist versenkt, also nehmen wir ihn aus der DotCom-Liste und steigen dann aus der Schleife aus.

Dem Benutzer das Ergebnis ausgeben.

Eine Nachricht ausgeben, die dem Benutzer sagt, wie gut er sich im Spiel angestellt hat.

Das Spielobjekt erstellen.

Dem Spielobjekt sagen, dass es das Spiel einrichten soll.

Dem Spielobjekt sagen, dass es die Haupt-Spielschleife starten soll (die die Benutzereingaben abfragt und die Rateversuche prüft).

Vorcode | Testcode | Hauptcode

# Die endgültige Version der DotCom-Klasse

```
import java.util.*;

public class DotCom {
   private ArrayList<String> zellorte;
   private String name;

   public void setZellorte(ArrayList<String> orte) {
      zellorte = orte;
   }

   public void setName(String n) {
      name = n;
   }

   public String prüfDich(String benutzereingabe) {
      String ergebnis = "Vorbei";
      int index = zellorte.indexOf(benutzereingabe);
      if (index >= 0) {
         zellorte.remove(index);

         if (zellorte.isEmpty()) {
            ergebnis = "Versenkt";
            System.out.println("Grrr! Sie haben " + name + " versenkt : ( ");
         } else {
            ergebnis = "Treffer";
         }  // if schließen
      } // if schließen
      return ergebnis;

   } // Methode schließen
} // Klasse schließen
```

Die DotCom-Instanzvariablen:
- eine ArrayList mit Zellorten
- der DotCom-Name

Eine Setter-Methode, die den Ort des Dot-Com aktualisiert (der zufällige Ort, der von der platziereDotCom()-Methode von Spielhelfer geliefert wird).

Ihre einfache Setter-Methode.

Die indexOf( )-Methode von ArrayList in Aktion! Wenn der Rateversuch einem der Einträge in der ArrayList entspricht, liefert indexOf( ) den Ort in der ArrayList zurück. Entspricht er keinem Ort, liefert indexOf( ) den Wert -1 zurück.

Die remove( )-Methode von ArrayList verwenden, um einen Eintrag zu löschen.

Die Methode isEmpty( ) verwenden, um zu prüfen, ob alle Orte erraten wurden.

Dem Benutzer mitteilen, wenn ein Dot-Com versenkt wurde.

>>Vorbei<< oder >>Treffer<< oder >>Versenkt<< zurückliefern.

# Boolesche Ausdrücke mit Super-Power

Wenn wir für unsere Schleifen oder if-Tests Boolesche Ausdrücke verwendet haben, waren diese bisher ziemlich einfach. In einigen der Code-Fertiggerichte, die Sie noch sehen werden, verwenden wir mächtigere Boolesche Ausdrücke. Und auch wenn wir davon ausgehen, dass Sie nicht vorblättern, denken wir, dass jetzt ein guter Zeitpunkt gekommen wäre, darüber zu reden, wie Sie Ihre Ausdrücke mit mehr Kraft ausstatten.

## Und- und Oder-Operatoren ( &&, || )

Nehmen wir an, Sie schreiben eine wähleKamera( )-Methode mit einer Menge Regeln zur Auswahl der Kamera. Vielleicht kann man zwischen Kameras wählen, deren Preis zwischen 50 und 1000 € liegt, und Sie möchten die Preisspanne in einigen Fällen auf einen kleineren Bereich begrenzen. Dann müssten Sie in etwa Folgendes sagen:

»Wenn der Preis im *Bereich* zwischen 300 € ***und*** 400 € liegt, wähle X.«

```
if (preis >= 300 && preis < 400) {
   kamera = "X";
}
```

Nehmen wir weiter an, Sie haben irgendeine Regel, die nur für *ein paar* der zehn verfügbaren Kameramarken gültig ist:

```
if (marke.equals("A") || marke.equals("B") )
{
   // mach Zeug nur für Marke A oder Marke B
}
```

Boolesche Ausdrücke können richtig groß und kompliziert werden:

```
if ((zoomTyp.equals("optisch") &&
   (zoomFaktor >= 3 && zoomFaktor <= 8)) ||
   (zoomTyp.equals("digital") &&
   (zoomFaktor >= 5 && zoomFaktor <= 12))) {
   // mach entsprechendes Zoom-Zeug
}
```

Wenn Sie das *richtig* technisch angehen, fragen Sie sich vielleicht, in welcher Reihenfolge diese Operatoren ausgewertet werden. Damit Sie nicht zu einem Experten in der geheimen Welt des Operatorenvorrangs werden müssen, empfehlen wir Ihnen, ***Klammern zu verwenden,*** um Ihren Code klarer zu machen.

## Nicht gleich ( != und ! )

Nehmen wir an, Sie haben eine Regel wie »eine bestimmte Sache ist für alle zehn Kameramodelle *wahr außer einem*.«

```
if (modell != 2000) {
   // mache Dinge, die nicht für Modell
   //2000 gedacht sind
}
```

Oder, wenn Sie Objekte wie Strings vergleichen möchten ...

```
if (!marke.equals("X")) {
   // mache Dinge, die nicht für Marke X
   //gelten
}
```

## Kurzschluss-Operatoren ( && , || )

Die Operatoren, die wir bisher betrachtet haben, && und ||, sind als ***Kurzschluss***-Operatoren bekannt. Bei && ist der Ausdruck nur dann wahr, wenn *beide* Seiten des && wahr sind. Sieht die JVM, dass die linke Seite eines &&-Ausdrucks false ist, hört sie deswegen bereits an dieser Stelle auf! Sie macht sich überhaupt nicht die Mühe, die rechte Seite anzusehen.

Bei || ist der Ausdruck schon wahr, wenn *eine* der Seiten wahr ist. Wenn die JVM sieht, dass die linke Seite true ist, erklärt sie deswegen hier den ganzen Ausdruck für wahr und spart sich, die rechte Seite zu prüfen.

Was ist so toll daran? Nehmen wir an, Sie haben eine Referenzvariable und sind sich nicht sicher, ob ihr ein Objekt zugewiesen wurde. Würden Sie versuchen, auf dieser null-Referenzvariablen (d.h., der kein Objekt zugewiesen ist) eine Methode aufzurufen, würden Sie eine NullPointerException erhalten. Machen Sie es doch lieber so:

```
if (refVar != null &&
   refVar.isGültigerTyp() ) {
   // mach 'Hab einen gültigen Typen'-Zeug
}
```

## Nicht-Kurzschluss-Operatoren ( & , | )

Wenn sie in Booleschen Ausdrücken verwendet werden, verhalten sich die Operatoren & und | wie ihre Gegenstücke && und ||. Der einzige Unterschied besteht darin, dass sie die JVM dazu zwingen, *immer beide* Seiten des Ausdrucks zu prüfen. Normalerweise werden & und | aber in einem anderen Kontext verwendet – zur Manipulation von Bits.

## Code-Fertiggericht

Das ist die Helferklasse für das Spiel. Neben der Methode für die Benutzereingabe (die eine Aufforderung für den Benutzer ausgibt und die Eingabe von der Kommandozeile liest) ist der besondere Service des Helfers, Zellorte für Dot-Coms zu erstellen. Wenn wir Sie wären, würden wir uns ganz langsam von diesem Code zurückziehen, nachdem Sie ihn eingegeben und kompiliert haben. Wir haben versucht, ihn ziemlich kompakt zu halten, damit Sie nicht so viel tippen müssen. Das hat allerdings dazu geführt, dass große Teile des Codes nicht besonders lesbar sind. Und denken Sie daran, dass Sie die Spielklasse DotComVersenken erst kompilieren können, wenn Sie diese Klasse haben.

```
import java.io.*;
import java.util.*;

public class SpielHelfer {

  private static final String alphabet = "abcdefg";
  private int rasterLänge = 7;
  private int rasterGröße = 49;
  private int [] raster = new int[rasterGröße];
  private int dotComAnzahl = 0;

  public String getBenutzereingabe(String prompt) {
     String eingabeZeile = null;
     System.out.print(prompt + "  ");
     try {
       BufferedReader is = new BufferedReader(
        new InputStreamReader(System.in));
       eingabeZeile = is.readLine();
       if (eingabeZeile.length() == 0 )  return null;
     } catch (IOException e) {
       System.out.println("IOException: " + e);
     }
     return eingabeZeile.toLowerCase();
  }

  public ArrayList<String> platziereDotCom(int dotComGröße) {
     ArrayList<String> alphaZellen = new ArrayList<String>();  // enthält Koordinaten der Form f6
     String temp = null;                          // temporärer String für concat
     int [] koordinaten = new int[dotComGröße];   // aktuelle Koordinatenkandidaten
     int versuche = 0;                            // aktueller Versuchszähler
     boolean erfolg = false;                      // schalter = einen guten Ort gefunden?
     int ort = 0;                                 // aktuelle Startposition

     dotComAnzahl++;                              // n-tes zu platzierendes Dot-Com
     int inkrement = 1;                           // horizontales Inkrement setzen
     if ((dotComAnzahl % 2) == 1) {               // wenn ungerades Dot-Com (vertikal platzieren)
       inkrement = rasterLänge;                   // vertikales Inkrement setzen
     }

     while ( !erfolg & versuche++ < 200 ) {       // Haupt-Suchschleife
         ort = (int) (Math.random() * rasterGröße);   // zufälligen Startpunkt holen
         //System.out.print(" Versuche " + ort);
         int x = 0;                               // n-te Position in zu platzierendem Dot-Com
         erfolg = true;                           // von Erfolg ausgehen
         while (erfolg && x < dotComGröße) {      // nach angrenzenden freien Orten suchen
           if (raster[ort] == 0) {                // wenn noch nicht benutzt
```

Hinweis: Wenn Sie ein paar zusätzliche Punkte machen wollen, versuchen Sie mal, die auskommentierten System.out.print(ln)-Zeilen in der Methode platziereDotCom( ) zu aktivieren. Sie zeigen Ihnen die Methode bei der Arbeit und ermöglichen Ihnen das Schummeln, weil Sie Ihnen die Orte der Dot-Coms verraten. Aber Sie helfen Ihnen auch, die Methode zu testen.

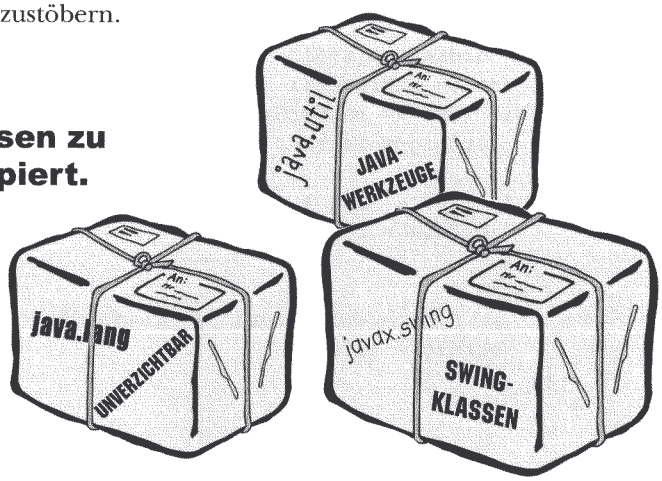
## SpielHelfer-Klasse (Fortsetzung)

```
            koordinaten[x++] = ort;                  // Ort speichern
            ort += inkrement;                        // 'nächste' angrenzende Zelle versuchen
            if (ort >= rasterGröße){                 // außerhalb der Grenzen - 'Ende'
              erfolg = false;                        // Fehlschlag
            }
            if (x>0 & (ort % rasterLänge == 0)) {    // außerhalb der Grenzen - rechter Rand
              erfolg = false;                        // Fehlschlag
            }
          } else {                                   // bereits belegten Ort gefunden
            // System.out.print(" verwendet " + ort);
            erfolg = false;                          // Fehlschlag
          }
        }
      }                                              // while-Ende

      int x = 0;                          // den Ort in alphabetische Koordinaten umwandeln
      int zeile = 0;
      int spalte = 0;
      // System.out.println("\n");
      while (x < dotComGröße) {
        raster[koordinaten[x]] = 1;       // die Punkte im Referenzraster als 'verwendet' markieren
        zeile = (int) (koordinaten[x] / rasterLänge);   // Zeilenwert holen
        spalte = koordinaten[x] % rasterLänge;          // numerischen Spaltenwert holen
        temp = String.valueOf(alphabet.charAt(spalte)); // in alphabetisches Zeichen umwandeln

        alphaZellen.add(temp.concat(Integer.toString(zeile)));
        x++;
       // System.out.print("  Koordinaten "+x+" = " + alphaZellen.get(x-1));

      }

      // System.out.println("\n");

      return alphaZellen;
    }
}
```

Diese Zeile sagt Ihnen genau, wo das Dot-Com platziert wurde.

# Die Bibliothek verwenden (die Java-API)

Dank der Hilfe von ArrayList haben Sie Ihren Weg durch das Spiel „Dot-Com versenken“ gemacht. Und wie versprochen ist es jetzt an der Zeit, etwas in der Java-Bibliothek herumzustöbern.




**In der Java-API sind Klassen zu Packages (Paketen) gruppiert.**

**Wenn Sie eine Klasse in der API verwenden möchten, müssen Sie wissen, in welchem Paket die Klasse steckt.**

Jede Klasse in der Java-Bibliothek gehört zu einem Paket. Das Paket hat einen Namen wie **javax.swing** (ein Package, das einige der Swing GUI-Klassen enthält, die Sie bald kennen lernen werden). ArrayList befindet sich im Package namens **java.util**, die, Überraschung, einen Haufen von *Hilfsklassen* enthält. In Kapitel 17 werden Sie noch viel mehr über Packages erfahren, auch wie Sie Ihre *eigenen* Klassen in Ihre *eigenen* Pakete packen. Jetzt sehen wir uns erst einmal an, wie man einige der Klassen *verwendet*, die mit Java ausgeliefert werden.

Eine Klasse aus der API in Ihrem eigenen Code zu verwenden ist einfach. Behandeln Sie die Klasse einfach, als hätten Sie sie selbst geschrieben ... als hätten Sie sie kompiliert und nun säße sie da und warte nur darauf, dass Sie sie verwenden. Es gibt allerdings einen großen Unterschied: An irgendeiner Stelle Ihres Codes müssen Sie den *vollständigen* Namen der Bibliotheksklasse angeben, die Sie verwenden möchten, und das bedeutet Paket-Name plus Klassenname.

Selbst wenn Sie sich dessen nicht bewusst waren, ***Sie haben bereits Klassen aus einem Package verwendet.*** System (System.out.println), String und Math (Math.random()) gehören alle zum Package **java.lang**.

## Sie müssen den vollständigen Namen* der Klasse kennen, die Sie in Ihrem Code verwenden möchten.

ArrayList ist nicht der *vollständige* Name von ArrayList, genau wie »Katharina« kein vollständiger Name ist (es sei denn, es ist ein Name wie Madonna oder Cher, aber das wollen wir jetzt nicht weiter vertiefen). Der vollständige Name von ArrayList ist eigentlich:

```
java.util.ArrayList
```

Package-Name

Klassenname

### Sie müssen Java sagen, welche Klasse, hier welche ArrayList, Sie verwenden möchten. Dazu haben Sie zwei Möglichkeiten:

**A** **IMPORTIEREN**

Setzen Sie an den Anfang Ihrer Quellcode-Datei eine Import-Anweisung:

```
import java.util.ArrayList;
public class MeineKlasse {... }
```

**ODER**

**B** **TIPPEN**

Geben Sie überall in Ihrem Code den vollständigen Namen an. Jedes Mal, wenn Sie ihn verwenden. *Überall, wo* Sie ihn verwenden.

Wenn Sie deklarieren und/oder instantiieren:

```
java.util.ArrayList<Hund> liste = new java.util.ArrayList<Hund>();
```

Wenn Sie ihn als Argumenttyp verwenden:

```
public void go(java.util.ArrayList<Hund> liste) { }
```

Wenn Sie ihn als Rückgabetyp verwenden:

```
public java.util.ArrayList<Hund> foo() {...}
```

*Wenn sich die Klasse nicht im Package java.lang befindet.

### Es gibt keine Dummen Fragen

**F: Warum muss es einen vollständigen Namen geben? Ist das der einzige Zweck von Packages?**

A: Packages sind aus drei wesentlichen Gründen wichtig. Erstens helfen sie Ihnen, die allgemeine Struktur eines Projekts oder einer Bibliothek zu ordnen. Klassen liegen nicht einfach auf einem schrecklich großen Haufen, sondern sind zu Packages gruppiert, die bestimmte Arten von Funktionalitäten bündeln (GUI oder Datenstrukturen oder Datenbankzeug beispielsweise). Zweitens geben Ihnen Packages Namensräume, die Konflikte verhindern, wenn Sie und zwölf weitere Programmierer in Ihrer Firma sich entschließen, unterschiedlichen Klassen den gleichen Namen zu geben. Wenn Sie eine Klasse namens Set haben und ein anderer (einschließlich der Java-API) ebenfalls eine Klasse namens Set hat, brauchen Sie eine Möglichkeit, der JVM zu sagen, *welche* Set-Klasse Sie verwenden möchten. Drittens bieten Packages eine Sicherheitsebene, weil Sie den Code, den Sie schreiben, so einschränken können, dass nur andere Klassen aus demselben Package darauf zugreifen können. Das werden Sie alles in Kapitel 17 erfahren.

**F: Gut. Aber gehen wir noch mal zu der Namenskonflikt-Geschichte zurück. Wie soll da ein vollständiger Name wirklich helfen? Was hält zwei Leute davon ab, einer Klasse den gleichen Package-Namen zu geben?**

A: Java hat eine Namenskonvention, die normalerweise verhindert, dass so etwas passiert, solange sich Entwickler an diese Konvention halten. Auch das werden wir uns in Kapitel 16 ansehen.

## Wo kommt das »x« her?

## (Oder: Was bedeutet es, wenn ein Package-Name mit javax beginnt?)

In der ersten und zweiten Java-Version (1.02 und 1.1) waren alle Klassen, die mit Java ausgeliefert wurden (d.h. alle Klassen in der Standardbibliothek) in Packages, die mit ***java*** begannen. Natürlich gab es immer ***java.lang*** – das Package, das Sie nicht importieren müssen. Und es gab ***java.net, java.io, java.util*** (auch wenn es damals so was wie ArrayList noch nicht gab) und ein paar andere einschließlich des Packages ***java.awt***, das GUI-relevante Klassen enthielt.

Am Horizont zeichneten sich aber schon andere Packages ab, die nicht bei der Standardbibliothek dabei waren. Diese Klassen wurden als ***Erweiterungen*** bezeichnet, und diese gab es in zwei wichtigen Varianten: *Standarderweiterungen* und *Nicht-Standarderweiterungen*. Standarderweiterungen waren die, die Sun als offiziell betrachtete. Diese unterschieden sich von experimentellen Packages, Test- und Beta-Packages, die vielleicht eines Tages die Bühne betreten würden, vielleicht aber auch nicht.

Konventionsgemäß wurde bei den Namen der Standarderweiterungen ein »x« an den üblichen Package-Anfang ***java*** angehängt. Die Urmutter aller Standarderweiterungen war die Swing-Bibliothek. Sie umfasste mehrere Packages, die alle mit ***javax.swing*** begannen.

Aber Standarderweiterungen können zu erstrangigen, mit Java ausgelieferten, standardmäßigen Bibliotheks-Packages befördert werden. Und genau das ist irgendwann ab Version 1.2 (die dann auch zur ersten zu Java 2 geadelten Version wurde) mit Swing passiert.

»Toll«, haben alle gedacht (wir eingeschlossen). »Jetzt hat jeder, der Java hat, auch die Swing-Klassen, und wir müssen uns nicht mehr damit herumschlagen, wie wir diese Klassen bei unseren Endbenutzern installieren.«

Aber der Ärger schwelte unter der Oberfläche. Denn wenn Packages befördert werden, müssen sie NATÜRLICH auch mit ***java*** und nicht mit ***javax*** beginnen. Jeder WEISS, dass Packages in der Standardbibliothek dieses »x« nicht haben, sondern nur Erweiterungen. Unmittelbar (und wir meinen unmittelbar) bevor die endgültige Version 1.2 veröffentlicht wurde, änderte Sun also die Package-Namen und löschte (unter anderem) das »x«. Bücher waren gedruckt und ausgeliefert worden und stellten den Swing-Code mit den neuen Namen vor. Die Namenskonventionen waren intakt. In der Java-Welt war alles in Ordnung.

Außer bei 20.000 und ein paar kreischenden Entwicklern, denen plötzlich klar wurde, dass mit dieser simplen Namensänderung das Desaster hereinbrach! Ihr gesamter Swing-basierter Code musste geändert werden! Was für ein Horror! Denken Sie nur an all die Import-Anweisungen, die mit ***javax*** begannen …

Und auf die letzte Minute, als sie schon verzweifelten, überzeugten die Entwickler Sun doch noch, »die Konvention zu vergessen und unseren Code zu retten«. Der Rest ist Geschichte. Wenn Sie in der Bibliothek ein Package sehen, das mit ***javax*** beginnt, wissen Sie also, dass es sein Leben als Erweiterung begann und dann befördert wurde.

## Punkt für Punkt

- **ArrayList** ist eine Klasse in der Java-API.
- Wenn Sie etwas in eine ArrayList stecken möchten, verwenden Sie **add()**.
- Wenn Sie etwas aus einer ArrayList entfernen möchten, verwenden Sie **remove()**.
- Wenn Sie herausfinden möchten, wo (und ob) sich etwas in einer ArrayList befindet, verwenden Sie **indexOf()**.
- Wenn Sie herausfinden möchten, ob eine ArrayList leer ist, verwenden Sie **isEmpty()**.
- Wenn Sie die Größe (die Anzahl von Elementen) einer ArrayList abfragen möchten, verwenden Sie die ***Methode*** **size()**.
- Aber denken Sie daran, dass Sie, wenn Sie die **Länge** (die Anzahl von Elementen) in einem guten alten Array abfragen möchten, die length-**Variable** verwenden.
- Eine ArrayList **wird dynamisch** auf die erforderliche Größe vergrößert. Sie wächst, wenn ihr Objekte hinzugefügt werden, und sie **schrumpft**, wenn Objekte aus ihr entfernt werden.
- Den Typ einer ArrayList deklarieren Sie mit einem **Typparameter**, einem Typnamen in spitzen Klammern. Beispiel: ArrayList<Button> heißt, dass die ArrayList nur Objekte vom Typ Button (oder, wie Sie in den nächsten paar Kapiteln lernen werden, Unterklassen von Button) aufnehmen kann.
- Obwohl eine ArrayList Objekte und keine elementaren Typen aufnehmen kann, »wrappt« (und »unwrappt«) sie einen elementaren Typ automatisch, wenn Sie ihn in eine ArrayList stecken (oder aus einer herausholen), und packt statt des elementaren Typs dann dieses Objekt in die ArrayList. (Später in diesem Buch werden Sie mehr zu diesem Feature erfahren.)
- Klassen werden zu Packages gruppiert.
- Eine Klasse hat einen vollständigen Namen, der eine Kombination aus Package-Namen und Klassennamen ist. Die Klasse ArrayList ist eigentlich java.util.ArrayList.
- Um Klassen zu verwenden, die sich in einem anderen Package als java.lang befinden, müssen Sie Java den vollständigen Namen dieser Klasse mitteilen.
- Sie verwenden entweder eine Import-Anweisung am Anfang Ihres Quellcodes, oder Sie geben den vollständigen Namen an allen Stellen an, an denen Sie die Klasse in Ihrem Code verwenden.

## Es gibt keine Dummen Fragen

**F: Macht `import` meine Klasse größer? Kompiliert es die importierten Klassen oder Packages tatsächlich in *meinen* Code?**

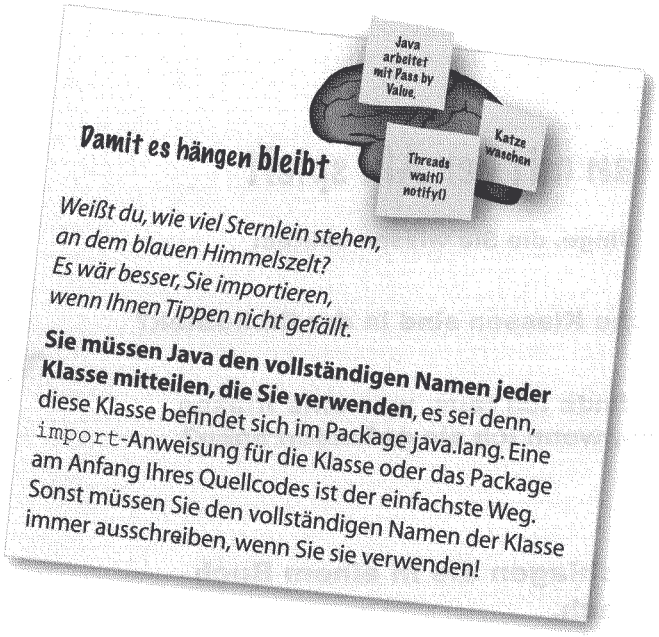
A: Sind Sie vielleicht C-Programmierer? Ein `import` ist nicht das Gleiche wie ein `include`. Die Antwort ist also nein und noch mal nein. Sprechen Sie mir nach: »Eine `import`-Anweisung spart mir Tipparbeit.« Und das ist es. Sie müssen sich keine Gedanken darüber machen, dass Ihr Code durch zu viele Imports aufgebläht oder langsamer wird. Ein `import` ist einfach die Art, auf die Sie Java den *vollständigen Namen einer Klasse mitteilen*.

**F: Okay. Aber warum muss ich die Klasse String nie importieren? Oder System?**

A: Erinnern Sie sich: Das Package java.lang erhalten Sie als »vorimportiert« kostenlos. Weil die Klassen in java.lang so grundlegend sind, müssen Sie für sie nicht die vollständigen Namen verwenden. Es gibt nur eine java.lang.String-Klasse und nur eine java.lang.System-Klasse, und Java weiß sehr gut, wo es die findet.

**F: Muss ich meine Klassen in ein Package stecken? Wie mach ich das? *Kann* ich das machen?**

A: In der harten Realität (die Sie einstweilen vermeiden sollten), ja, da *sollten* Sie Ihre Klassen in Packages packen. Das werden wir uns in Kapitel 17 ausgiebig ansehen. Im Augenblick packen wir unsere Codebeispiele nicht in ein Package.




**Noch einmal, für den unwahrscheinlichen Fall, dass es bei Ihnen noch nicht angekommen ist:**

*»Gut zu wissen, dass es im Package java.util eine ArrayList gibt. Aber wie hätte **ich** das allein rausfinden können?«*

- Julia, 31, Hand-Model

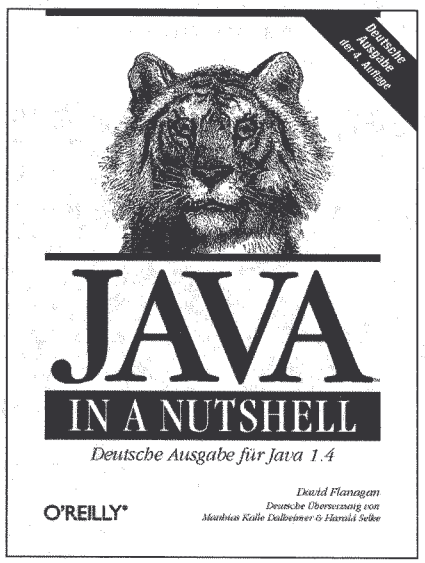
# Wie man mit der API spielt

**Zwei Dinge, die Sie wissen sollten:**

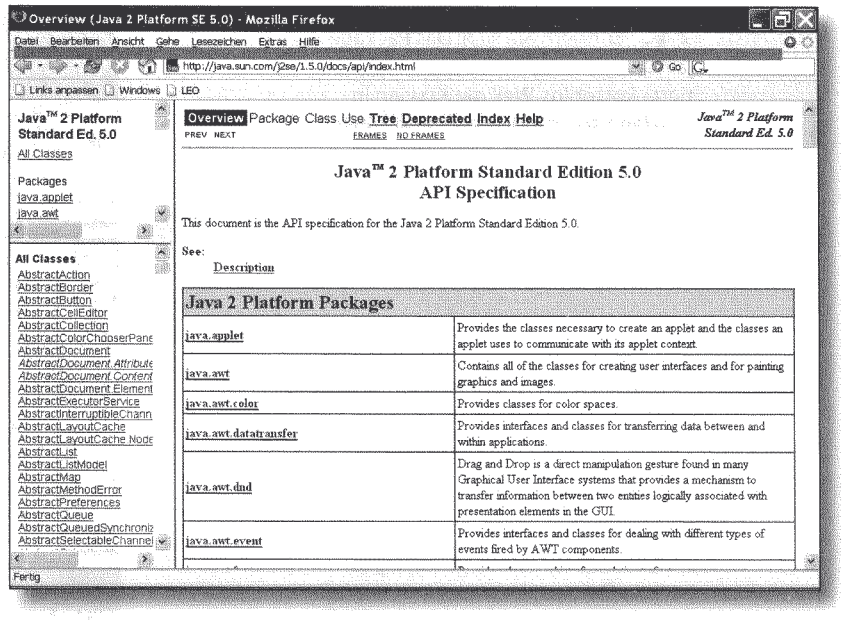
**1 Welche Klassen sind in der Bibliothek?**

**2 Wie finde ich raus, was eine Klasse kann, wenn ich sie gefunden habe?**

## 1 Schlagen Sie in einem Buch nach.




## 2 Nutzen Sie die HTML-API-Dokumentation.



## 1 Schlagen Sie in einem Buch nach.

Ein Referenzbuch durchzublättern ist der beste Weg, um herauszufinden, was es in der Java-Bibliothek gibt. Beim Durchblättern können Sie schnell über eine Klasse stolpern, die nützlich aussieht.

Klassenname

Package-Name

Klassenbeschreibung

**AbstractList** **Java 1.2**

java.util *Collection*

Diese abstrakte Klasse ist eine partielle Implementierung des Interface List, mit der es einfach ist, eigene List-Implementierungen auf der Basis von List-Elementen mit wahlfreiem Zugriff (wie etwa in einem Array gespeicherten Objekten) zu implementieren. Wenn Ihre List-Implementierung auf sequentiellem Datenzugriff (wie etwa in einer verketteten Liste) basieren soll, dann verwenden Sie statt dessen AbstractSequentialList als Basisklasse.

Um eine nicht-modifizierbare List-Implementierung zu schreiben, leiten Sie einfach von AbstractList ab und überschreiben die (geerbten) Methoden size() und get(). Um eine modifizierbare Liste zu programmieren, müssen Sie außerdem set() sowie optional add() und remove() überschreiben. Diese drei Methoden sind optional; wenn Sie sie nicht selbst implementieren, lösen sie einfach eine UnsupportedOperationException-Ausnahme aus. Alle anderen Methoden des Interface List werden mit Hilfe von size(), get(), add() und remove() implementiert. Manchmal kann es aus Performance-Gründen aber auch sinnvoll sein, weitere Methoden zu überschreiben. Konventionsgemäß sollten alle Implementierungen von List zwei Konstruktoren definieren: einen argumentlosen und einen weiteren, der ein Collection-Element erwartet, das die initialen Elemente der Liste angibt.

```
public abstract class AbstractList extends AbstractCollection implements java.util.List {
// Protected-Konstruktoren
  protected AbstractList();
// Methoden, die List implementieren
  public boolean add(Object o);
  public void add(int index, Object element);
  public boolean addAll(int index, Collection c);
  public void clear();
  public boolean equals(Object o);
  public abstract Object get(int index);
  public int hashCode();
  public int indexOf(Object o);
  public Iterator iterator();
  public int lastIndexOf(Object o);
  public ListIterator listIterator();
  public ListIterator listIterator(int index);
  public Object remove(int index);
  public Object set(int index, Object element);
  public java.util.List subList(int fromIndex, int toIndex);
// Protected-Instanzmethoden
  protected void removeRange(int fromIndex, int toIndex);
// Protected-Instanzfelder
  protected transient int modCount;
}
```

Methoden (und andere Dinge, über die wir später sprechen werden)

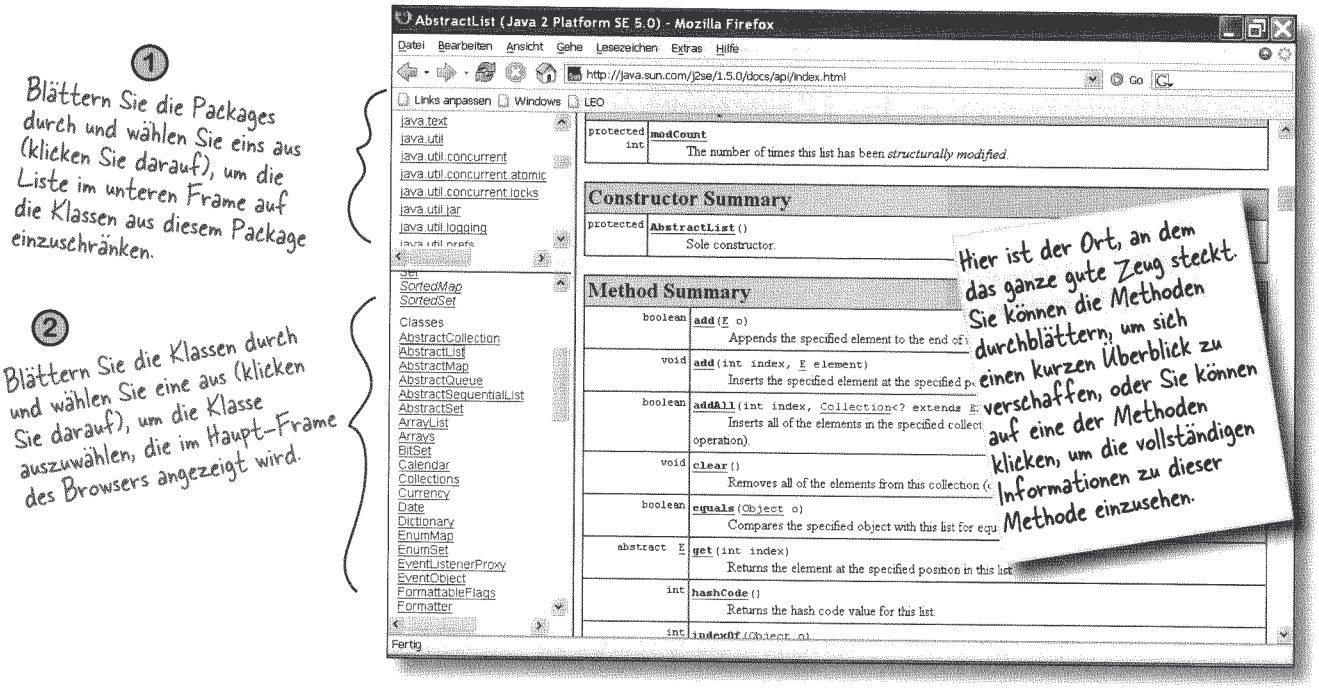
*Abgeleitete Klassen:* AbstractSequentialList, ArrayList, Vector

## 2 Nutzen Sie die HTML-API-Dokumentation

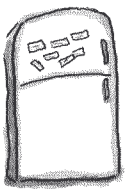
Java kommt mit einem wunderbaren Satz von Onlinedokumentationen, die, seltsamerweise, Java-API genannt werden. Sie sind Teil eines größeren Pakets, das als die Java 5 Standard Edition Documentation bezeichnet wird (je nach Wochentag kann es auch sein, dass Sun sie gerade mal wieder als »Java 2 Standard Edition 5.0« bezeichnet). Diese Dokumentation müssen Sie separat herunterladen, sie kommt nicht vakuumverpackt mit dem Java 5-Download. Wenn Sie eine Breitband-Internetverbindung oder sehr viel Geduld haben, können Sie sie auch unter java.sun.com durchblättern. Aber glauben Sie uns, Sie wollen sie sicher auf der Festplatte haben.

Die API-Dokumentation ist die beste Referenz, um mehr Informationen zu einer Klasse und ihren Methoden zu erhalten. Nehmen wir an, Sie haben einen Blick in ein Referenzbuch geworfen und in java.util eine Klasse namens Calendar gefunden. Das Buch sagt Ihnen ein bisschen zu der Klasse, das gerade ausreicht, um Ihnen klar zu machen, dass das genau das ist, was Sie brauchen. Aber jetzt müssen Sie immer noch etwas zu den Methoden erfahren.

Das Referenzbuch sagt Ihnen vielleicht, was die Methoden als Argumente erwarten und was sie zurückliefern. Sehen Sie sich beispielsweise ArrayList an. Im Referenzbuch finden Sie die Methode indexOf(), die wir in der Klasse DotCom verwendet haben. Aber alles, was Sie erfahren, ist, dass es eine Methode namens indexOf() gibt, die als Argument ein Objekt nimmt und den Index (ein int) für das Objekt zurückliefert. Aber eine entscheidende Sache wissen Sie noch nicht: Was passiert, wenn sich das Objekt nicht in der ArrayList befindet? Der Blick auf die Methodensignatur allein sagt Ihnen noch nicht, wie das funktioniert. Aber die API-Dokumentation sagt es Ihnen (meistens zumindest). Die API-Dokumentation sagt Ihnen, dass die Methode indexOf() den Wert –1 zurückliefert, wenn sich das Objekt nicht in der ArrayList befindet. Auf diese Weise hatten wir erfahren, dass wir die Methode gleichzeitig als Mittel verwenden konnten, um herauszufinden, ob sich ein Objekt überhaupt in der ArrayList befindet, und um den Index des Objekts abzurufen, wenn es sich darin befindet. Ohne die API-Dokumentation hätten wir vielleicht gedacht, dass die Methode indexOf() hochgehen würde, wenn sich das Objekt nicht in der ArrayList befindet.



# Code-Magneten



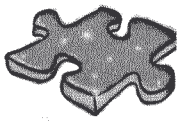
Können Sie die Codeschnipsel rekonstruieren, um ein funktionierendes Java-Programm zu erstellen, das die unten angezeigte Ausgabe hervorbringt? **Hinweis:** Um diese Übung durchzuführen, benötigen Sie eine NEUE Information – wenn Sie in die API für ArrayList blicken, finden Sie eine *zweite* add-Methode, die zwei Argumente erwartet:

**add(int index, Object o)**

Sie ermöglicht Ihnen, der ArrayList zu sagen, *wo* das Objekt eingefügt werden soll, das Sie hinzufügen.

```
a.remove(2);
printAL(a);
printAL(a);
a.add(0,"null");
a.add(1,"eins");
public static void printAL(ArrayList<String> al) {
if (a.contains("zwei")) {
   a.add("2.2");
}
a.add(2,"zwei");
public static void main (String[] args) {
System.out.print(element + "  ");
}
System.out.println(" ");
if (a.contains("drei")) {
   a.add("vier");
}
public class ArrayListMagnet {
if (a.indexOf("vier") != 4) {
    a.add(4, "4.2");
}
}
}
import java.util.*;
printAL(a);
ArrayList<String> a = new ArrayList<String>();
for (String element : al) {
a.add(3,"drei");
printAL(a);
```

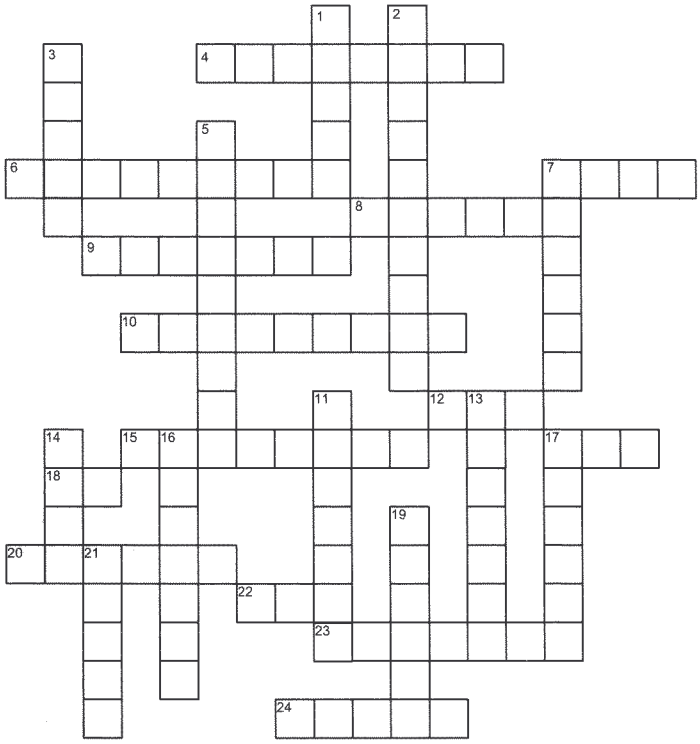
```
Datei Bearbeiten Fenster Hilfe Tanzen
% java ArrayListMagnet
null  eins  zwei  drei
null  eins  drei  vier
null  eins  drei  vier  4.2
null  eins  drei  vier  4.2
```



# JavaKreuz-wörträtsel 7.0

Wie Ihnen dieses Kreuzworträtsel hilft, Java zu lernen? Na, alle Wörter (abgesehen von einem Ablenkungsmanöver) **sind** Java-bezogen.

**Hinweis:** Wenn Sie sich unsicher sind, denken Sie an ArrayList.



## Waagerecht

4 Kehrt gerne an den Anfang zurück.

6 Gedoptes Array.

7 Größter Ganzzahl-Container.

8 Gleitkomma-Default.

9 Dürremethode.

10 Enthält irgendwas.

12 Akronym für Bibliothek.

15 Als ob ...

17 Kein Objekt.

18 Ursprung einer Abzweigung.

20 Liefert eine Wertkopie.

22 Lässt eine ArrayList wachsen.

23 Adressierbare Einheit.

24 Was Arrays und Einkaufen gemeinsam haben.

## Senkrecht

1 Mit gleitender Trennwand.

2 Wo Packages rumhängen.

3 Am zweitkleinsten.

5 Ist so _____, dass es kein Verhalten hat.

7 Das Gegenstück zu 14.

11 Wo die Java-Action spielt.

13 Größte Einheit in Bibliotheken.

14 Ausdehnung.

16 Wo steckt das Ding?

17 Anweisung für faule Finger.

19 Das, was entgegensteht.

21 Spanische Geek-Snacks (Hinweis: hat nichts mit Java zu tun).

## Weitere Hinweise:

Waagerecht
9 Denken Sie an ArrayList.
17 Üblicher elementarer Typ.
23 Denken Sie an ArrayList.
24 Er macht eine _____.

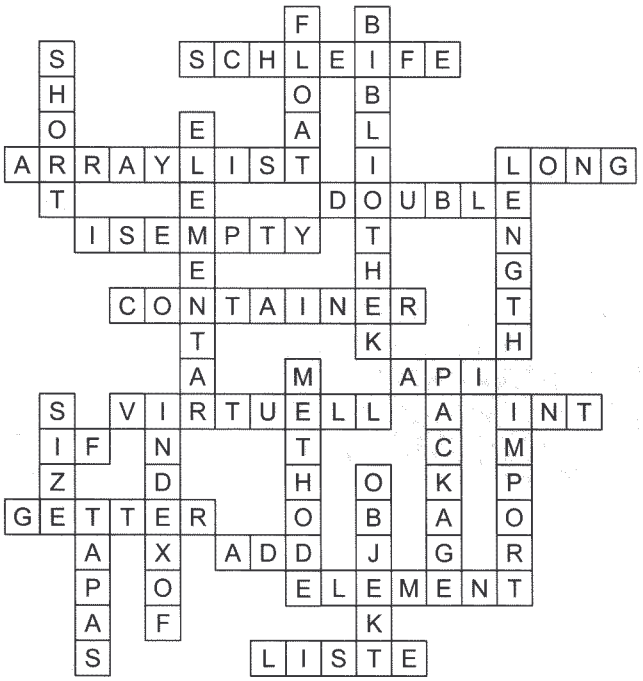
Senkrecht
5 Gibt es in 8 Arten.
7 Ausdehnung eines Arrays.
11 Was ist überschreibbar?
16 Denken Sie an ArrayList.

## Lösungen zu den Übungen

```
Datei Bearbeiten Fenster Hilfe Tanzen
% java ArrayListMagnet
null  eins  zwei  drei
null  eins  drei  vier
null  eins  drei  vier  4.2
null  eins  drei  vier  4.2
```

```
import java.util.*;
public class ArrayListMagnet {
  public static void main (String[] args) {
    ArrayList<String> a = new ArrayList<String>();
    a.add(0,"null");
    a.add(1,"eins");
    a.add(2,"zwei");
    a.add(3,"drei");
    printAL(a);
    if (a.contains("drei")) {
      a.add("vier");
    }
    a.remove(2);
    printAL(a);
    if (a.indexOf("vier") != 4) {
      a.add(4, "4.2");
    }
    printAL(a);
    if (a.contains("zwei")) {
      a.add("2.2");
    }
    printAL(a);
  }
  public static void printAL(ArrayList<String> al) {
    for (String element : al) {
      System.out.print(element + "  ");
    }
    System.out.println(" ");
  }
}
```

# JavaKreuzworträtsel-Antworten



## Spitzen Sie Ihren Bleistift

Schreiben Sie Ihren EIGENEN Satz mit Hinweisen! Sehen Sie sich jedes Wort an und versuchen Sie, Ihre eigenen Hinweise zu schreiben. Versuchen Sie sie einfacher oder schwerer oder technischer zu machen als die, die wir Ihnen angeboten haben.

### Waagerecht

4 _____

6 _____

7 _____

8 _____

9 _____

10 _____

12 _____

15 _____

17 _____

18 _____

20 _____

22 _____

23 _____

24 _____

### Senkrecht

1 _____

2 _____

3 _____

5 _____

7 _____

11 _____

13 _____

14 _____

16 _____

17 _____

19 _____

21 _____

# Besser leben in Objekthausen

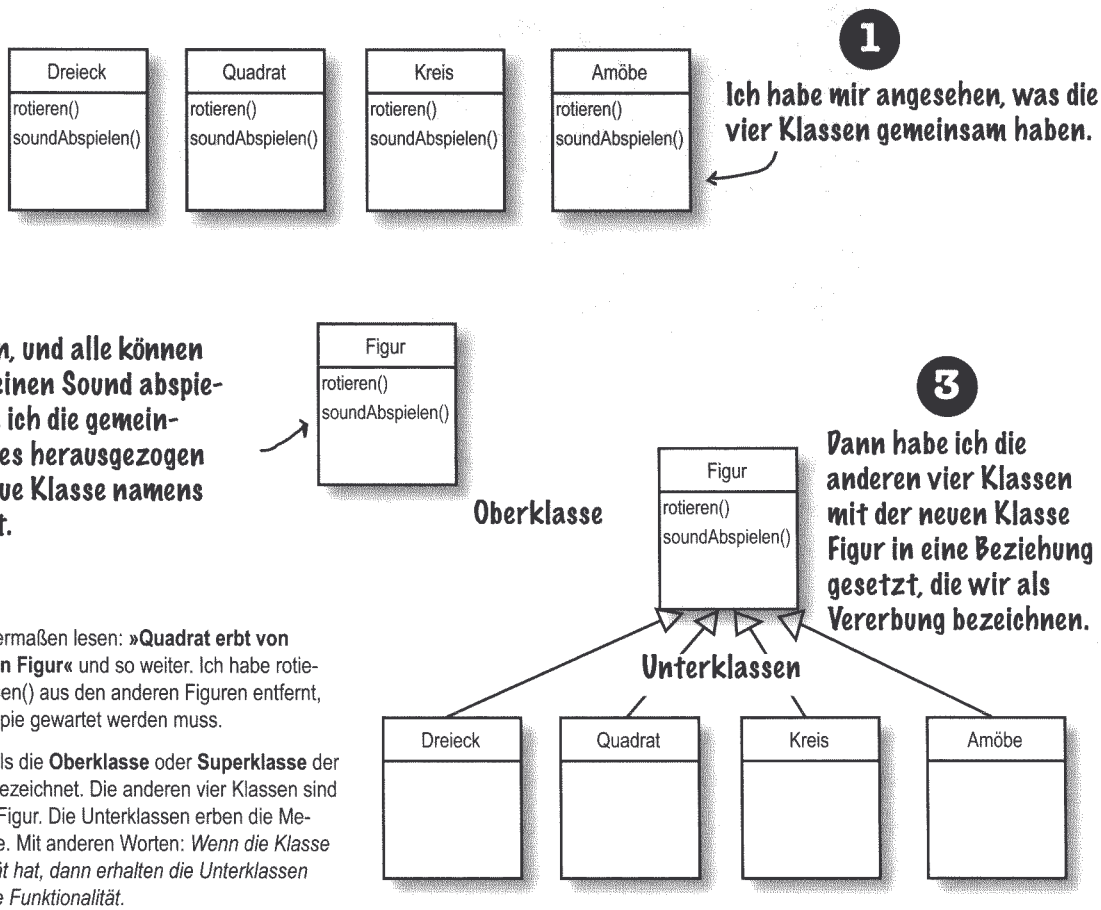
**Planen Sie Ihre Programme mit Blick auf die Zukunft.** Wenn es eine Möglichkeit gäbe, Java-Code so zu schreiben, dass Sie mehr Urlaub machen könnten, was wäre die Ihnen wert? Was wäre, wenn Sie Code schrieben, den ein *anderer* **problemlos** erweitern könnte? Und was wäre, wenn Ihr Code flexibel genug für diese verfluchten Spezifikationsänderungen in letzter Minute wäre? Würde Sie das nicht brennend interessieren? Dann ist heute Ihr Glückstag. Alles das bieten wir Ihnen für nur dreimal schlappe 60 Minuten Aufwand. Wenn Sie auf den Polymorphie-Zug springen, lernen Sie die fünf Schritte zu besserem Klassen-Design, 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. Zögern Sie nicht, ein derart gutes Angebot gibt Ihnen die Freiheit und Programmierflexibilität, die Sie verdienen. Es geht schnell, es ist einfach, und Sie können es jetzt haben. Beginnen Sie noch heute, und wir legen als Zugabe ein wenig Abstraktion drauf!

# Noch einmal zu den Stuhlkriegen ...

*Erinnern Sie sich an Kapitel 2, als Harry (der prozedurale Typ) und Stefan (der OO-Typ) um den Designerstuhl kämpften? Sehen wir uns noch mal ein paar Teile der Geschichte an, um uns die Grundlagen der Vererbung ins Gedächtnis zu rufen.*

**HARRY**: Du hast da doppelten Code! Die Prozedur rotieren kommt in allen vier Figur-Dingern vor. Das ist ein dämliches Design. Wie kann so was jemals gut sein?

**STEFAN**: Oh. Ich vermute, du hast den endgültigen Entwurf nicht gesehen. Komm, ich erklär dir, wie **OO-Vererbung** funktioniert, Harry.




Das kannst du folgendermaßen lesen: **»Quadrat erbt von Figur«**, **»Kreis erbt von Figur«** und so weiter. Ich habe rotieren() und soundAbspielen() aus den anderen Figuren entfernt, damit nur noch eine Kopie gewartet werden muss.

Die Klasse Figur wird als die **Oberklasse** oder **Superklasse** der vier anderen Klassen bezeichnet. Die anderen vier Klassen sind die **Unterklassen** von Figur. Die Unterklassen erben die Methoden der Superklasse. Mit anderen Worten: *Wenn die Klasse Figur eine Funktionalität hat, dann erhalten die Unterklassen automatisch die gleiche Funktionalität.*

## Was ist mit dem rotieren() von Amöbe?

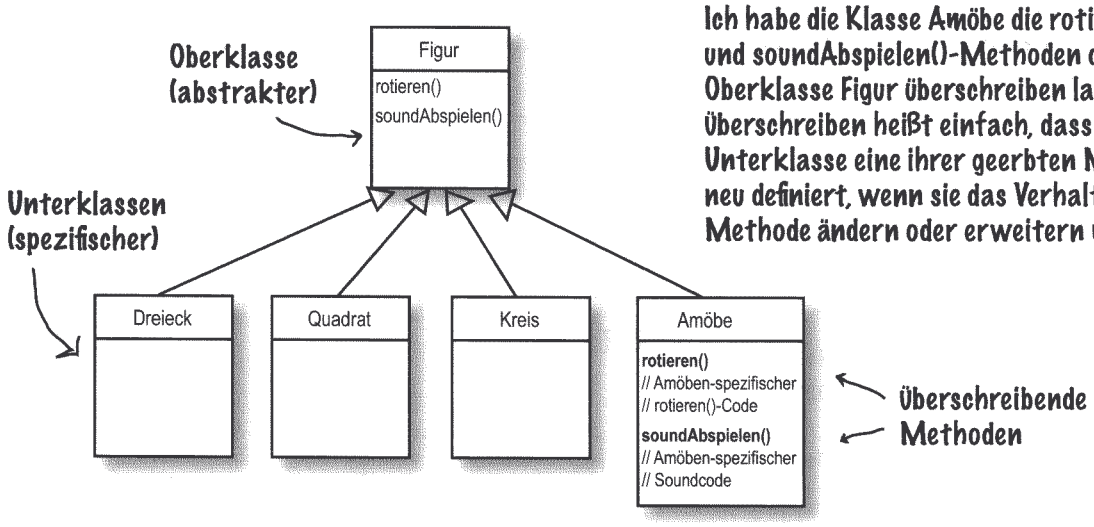
**HARRY:** War das nicht gerade das Problem, das wir hatten – dass die Figur Amöbe ganz andere rotieren- und soundAbspielen-Prozeduren hat?

Wie kann Amöbe etwas anderes machen, wenn sie ihre Funktionalität von der Klasse Figur *erbt*?

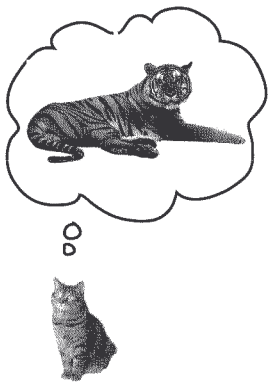
**STEFAN:** Das ist der letzte Schritt. Die Klasse Amöbe ***überschreibt*** die Methoden der Klasse Figur. Zur Laufzeit weiß die JVM dann genau, welche *rotieren*()-Methode ausgeführt werden soll, wenn irgendwer der Amöbe sagt, dass sie rotieren soll.

**4**

**Ich habe die Klasse Amöbe die rotieren()- und soundAbspielen()-Methoden der Oberklasse Figur überschreiben lassen. Überschreiben heißt einfach, dass eine Unterklasse eine ihrer geerbten Methoden neu definiert, wenn sie das Verhalten dieser Methode ändern oder erweitern muss.**




---



## KOPF-NUSS

Wie würden Sie eine Hauskatze und einen Tiger in einer Vererbungsstruktur darstellen? Ist eine domestizierte Katze eine spezielle Version eines Tigers? Was wäre die Superklasse, und was wäre die Unterklasse? Oder sind sie beide Unterklassen einer *anderen* Klasse?

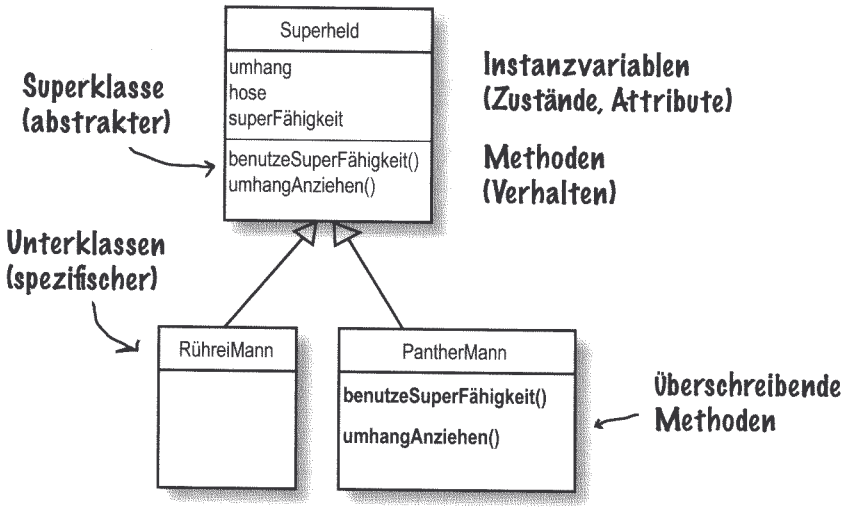
Wie würden Sie eine Vererbungsstruktur entwerfen? Welche Methoden müssten überschrieben werden?

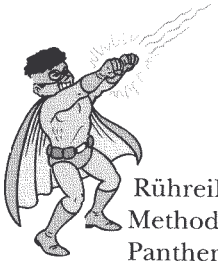
Denken Sie darüber nach – *bevor* Sie umblättern.

# Vererbung verstehen

Wenn Sie mit Hilfe von Vererbung entwerfen, stecken Sie Ihren allgemeinen Code in eine Klasse und sagen dann den spezifischeren Klassen, dass die allgemeine (abstraktere) Klasse ihre Superklasse ist. Wenn eine Klasse von einer anderen Klasse erbt, **erbt die Unterklasse von der Oberklasse.**

In Java sagen wir, dass die **Unterklasse die Superklasse *erweitert***. Eine Vererbungsbeziehung bedeutet, dass die Unterklasse die **Member** der Superklasse erbt. Wenn wir von »Membern einer Klasse« sprechen, meinen wir die Instanzvariablen und Methoden. Wenn PantherMann eine Unterklasse von Superheld ist, erbt PantherMann beispielsweise automatisch die Instanzvariablen und Methoden, die alle Superhelden gemeinsam haben, einschließlich `umhang`, `hose`, `superFähigkeit`, `benutzeSuperFähigkeit()` und so weiter. Aber die PantherMann-**Unterklasse kann eigene neue Methoden und Instanzvariablen hinzufügen** und **die Methoden überschreiben, die sie von der Superklasse** Superheld **erbt.**



RühreiMann braucht kein eigenes Verhalten, also überschreibt er keine Methoden. Die Methoden und Variablen in Superheld reichen aus. Aber PantherMann stellt besondere Anforderungen an seinen Umhang und seine Super-Fähigkeiten, also werden von der Klasse PantherMann die beiden Methoden `benutzeSuperFähigkeit()` und `umhangAnziehen()` überschrieben.

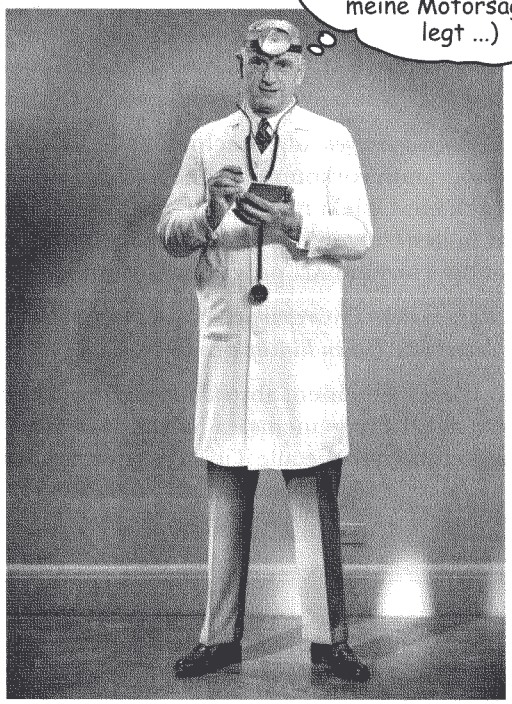
**Instanzvariablen werden nicht überschrieben,** weil sie nicht überschrieben werden müssen. Sie definieren kein besonderes Verhalten, also kann eine Unterklasse einer geerbten Instanzvariablen die Werte geben, die sie möchte. PantherMann kann die geerbte Variable `hose` auf Pink setzen, RühreiMann seine auf Weiß.

## Ein Vererbungsbeispiel:

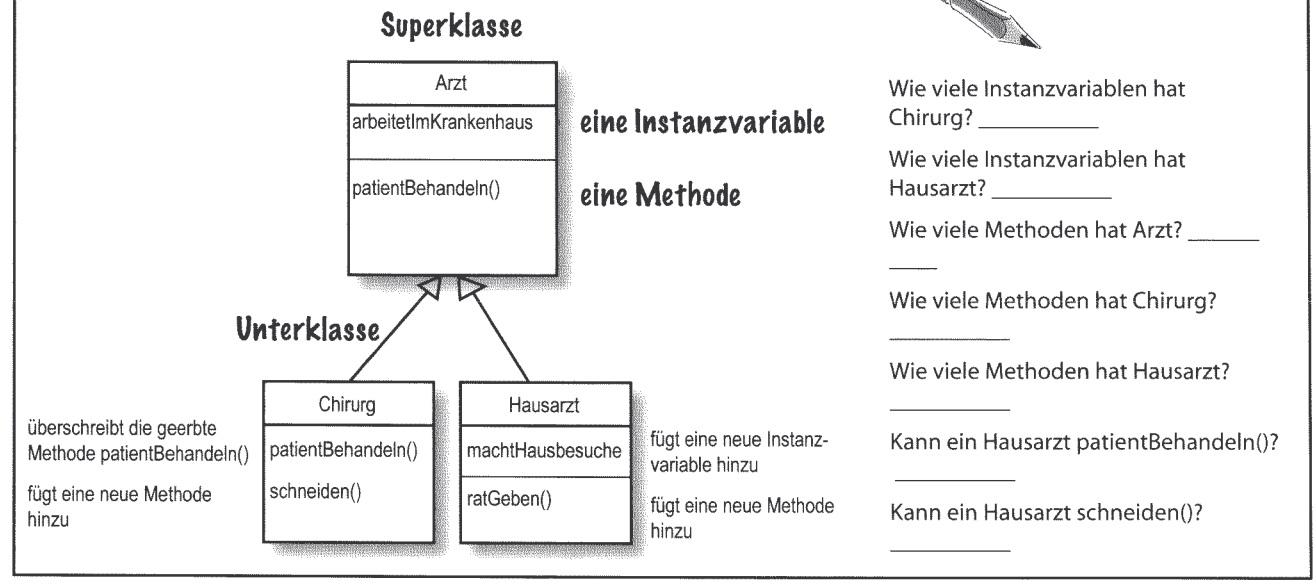
```
public class Arzt {

  boolean arbeitetImKrankenhaus;

  void patientBehandeln() {
    // Patient untersuchen
  }
}
```

```
public class Hausarzt extends Arzt {

  boolean machtHausbesuche;
  void ratGeben() {
    // Ratschlag geben
  }
}
```

```
public class Chirurg extends Arzt {

  void patientBehandeln() {
    // Operation durchführen
  }

  void schneiden(){
    // Schnitt durchführen (Autsch!)
  }
}
```




### Spitzen Sie Ihren Bleistift




Wie viele Instanzvariablen hat Chirurg? _____

Wie viele Instanzvariablen hat Hausarzt? _____

Wie viele Methoden hat Arzt? _____

Wie viele Methoden hat Chirurg? _____

Wie viele Methoden hat Hausarzt? _____

Kann ein Hausarzt patientBehandeln()? _____

Kann ein Hausarzt schneiden()? _____

# Entwerfen wir einen Vererbungsbaum für eine Tier-Simulation

Stellen Sie sich vor, Sie werden gebeten, ein Simulationsprogramm zu schreiben, in dem der Benutzer eine Vielzahl verschiedener Tiere in eine Umgebung stecken kann, um zu beobachten, was passiert. Wir müssen die Sache jetzt hier nicht programmieren. Uns interessiert vor allem das Design.

Uns wurde eine Liste mit *einigen,* aber nicht mit allen Tieren gegeben, die im Programm vorkommen sollen. Wir wissen, dass jedes Tier durch ein Objekt repräsentiert wird und dass sich dieses Objekt in der Umgebung herumbewegt und dort das macht, worauf die einzelne Art programmiert wurde.

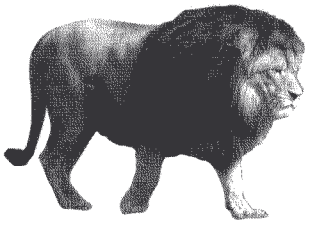
***Und wir möchten, dass andere Programmierer dem Programm jederzeit beliebige Arten von Tieren hinzufügen können.***

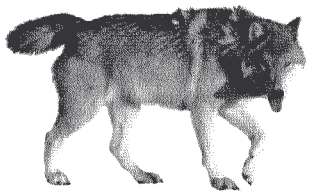
Zuerst müssen wir die allgemeinen, abstrakten Kennzeichen herausarbeiten, die alle Tiere gemeinsam haben, und diese dann in eine Klasse einbauen, die alle Tier-Klassen erweitern können.

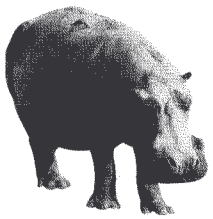
## 1 Nach Objekten suchen, die Attribute und Verhalten gemeinsam haben.

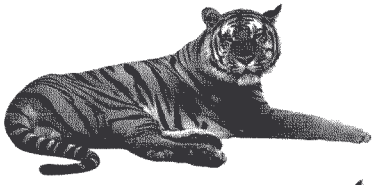
**Was haben diese sechs Typen gemeinsam? Das hilft Ihnen, Verhalten herauszuziehen. (Schritt 2)**

**Wie sind diese Typen verwandt? Das hilft Ihnen, die Beziehungen im Vererbungsbaum zu definieren. (Schritte 4 und 5)**









# Mit Vererbung Codeverdopplung in Unterklassen verhindern

Wir haben fünf ***Instanzvariablen***:

***bild*** – Der Dateiname des JPEG für dieses Tier.

***nahrung*** – Der Typ Nahrung, den dieses Tier isst. Kann zurzeit nur zwei Werte annehmen: *Fleisch* oder *Gras*.

***hunger*** – Ein int, das die Größe des Hungers des Tiers repräsentiert. Es ändert sich in Abhängigkeit davon, wann (und wie viel) das Tier isst.

***grenzen*** – Werte, die die Größe und Breite des Raums (beispielsweise 640 x 480) repräsentieren, in dem sich das Tier bewegt.

***ort*** – Die x- und y-Koordinaten, an denen sich das Tier im Raum befindet.

Wir haben vier ***Methoden***:

***geräuschMachen***() – Verhalten, wenn das Tier ein Geräusch machen soll.

***essen()*** – Verhalten, wenn das Tier auf seine bevorzugte Nahrung stößt: *Fleisch* oder *Gras*.

***schlafen()*** – Verhalten, wenn das Tier als schlafend betrachtet wird.

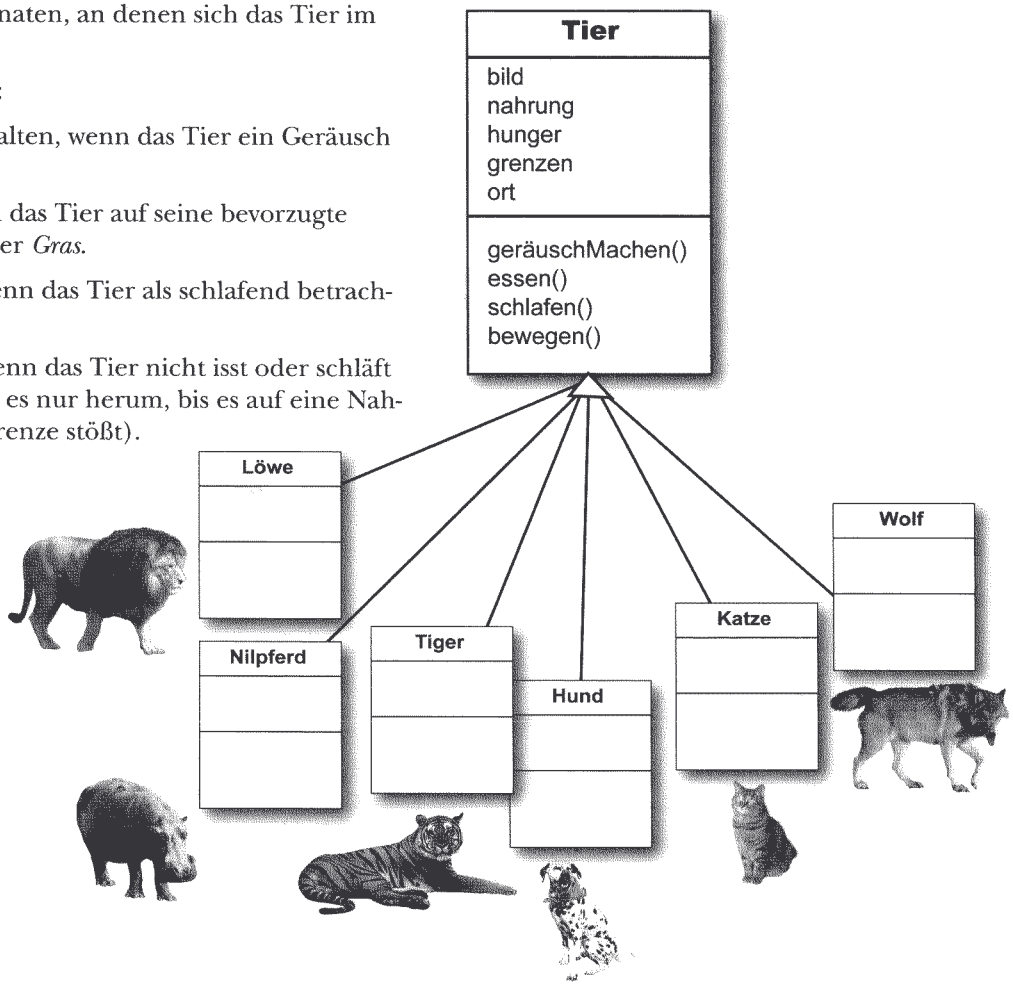
***bewegen()*** – Verhalten, wenn das Tier nicht isst oder schläft (wahrscheinlich wandert es nur herum, bis es auf eine Nahrungsquelle oder eine Grenze stößt).

**2**

Eine Klasse entwerfen, die den allgemeinen Zustand und das allgemeine Verhalten repräsentiert.

**All diese Objekte sind Tiere, also machen wir eine Superklasse, die wir Tier nennen.**

**In sie packen wir alle Methoden und Instanzvariablen, die Tiere brauchen können.**



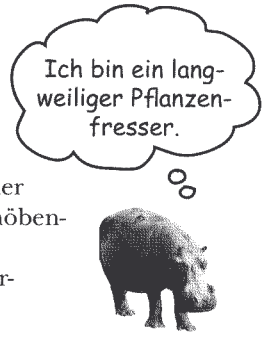
# Essen alle Tiere auf gleiche Weise?

Über eine Sache sind wir uns doch einig: Die Instanzvariablen sind für *alle* Tierarten geeignet. Ein Löwe hat einen eigenen Wert für bild, nahrung (Fleisch höchstwahrscheinlich), hunger, grenzen und ort. Ein Nilpferd hat andere *Werte* für seine Instanzvariablen, aber es hat trotzdem die gleichen Variablen, die auch die anderen Tier-Typen haben. Das Gleiche gilt für Hund, Tiger und so weiter. Aber was ist mit dem *Verhalten?*

## Welche Methoden sollten wir überschreiben?

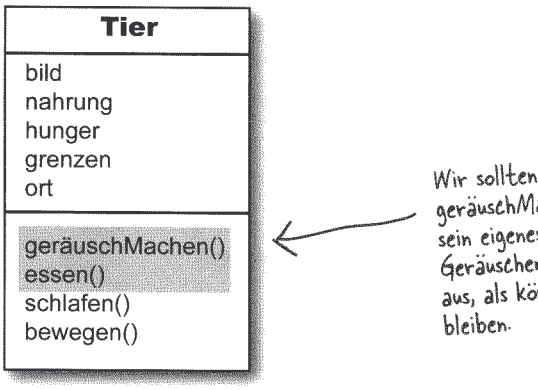
Macht ein Löwe das gleiche **Geräusch** wie ein Hund? **Isst** eine Katze wie ein Nilpferd? Vielleicht in *Ihrer* Version, aber in unserer sind die Art zu essen und die Geräusche, die sie machen, tierspezifisch. Wir können nicht herausfinden, wie wir diese Methoden so schreiben, dass sie für alle Tiere funktionieren. Okay, das ist nicht wahr. Wir könnten die Methode geräuschMachen() beispielsweise so schreiben, dass sie nur eine Sounddatei abspielt, die für den Typ in einer Instanzvariablen definiert wird, ansonsten aber nicht spezialisiert ist. Einige Tiere machen eventuell verschiedene Geräusche in unterschiedlichen Situationen (beispielsweise eins beim Essen und ein anderes, wenn sie auf einen Feind stoßen usw.).

Was wir bei der Amöbe gemacht haben, die die rotieren()-Methode der Klasse Figur überschrieb, um ein Amöben-spezifischeres Verhalten zu erhalten, müssen wir hier dann für unsere Tier-Unterklassen machen.



**3** Entscheiden, ob eine Unterklasse Verhalten (Methodenimplementierungen) braucht, das für diesen bestimmten Typ Unterklasse spezifisch ist.

**Wir haben uns die Klasse Tier angesehen und entschieden, dass essen() und geräuschMachen() von den einzelnen Unterklassen überschrieben werden sollten.**




Wir sollten die beiden Methoden essen() und geräuschMachen() überschreiben, damit jedes Tier sein eigenes Verhalten für das Essen und das Geräuschemachen definieren kann. Noch sieht es so aus, als könnten schlafen() und bewegen() allgemein bleiben.

# Nach weiteren Vererbungsmöglichkeiten suchen

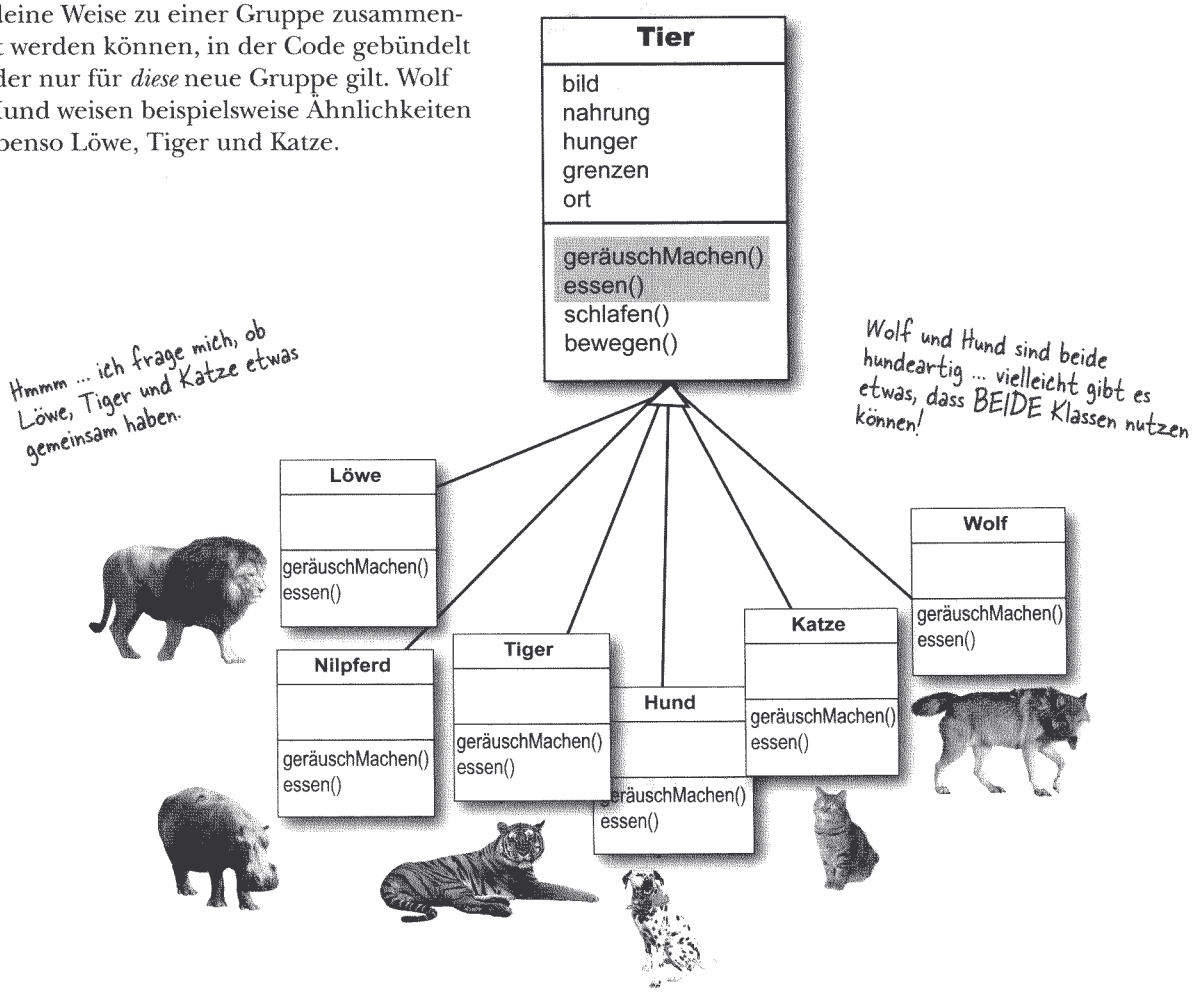
Die Klassenhierarchie nimmt langsam Gestalt an. Wir haben alle Unterklassen die Methoden *geräuschMachen()* und *essen()* überschreiben lassen, damit man das Bellen eines Hundes nicht mit dem Miauen einer Katze verwechseln kann (was schließlich für beide eine Beleidigung wäre). Ein Nilpferd isst dann auch nicht wie ein Löwe.

Aber vielleicht können wir noch mehr machen. Wir müssen die Unterklassen von Tier untersuchen, um zu prüfen, ob zwei oder mehr auf irgendeine Weise zu einer Gruppe zusammengefasst werden können, in der Code gebündelt wird, der nur für *diese* neue Gruppe gilt. Wolf und Hund weisen beispielsweise Ähnlichkeiten auf. Ebenso Löwe, Tiger und Katze.

**4**

Nach weiteren Möglichkeiten für Abstraktionen suchen, indem wir zwei oder mehr *Unterklassen* finden, die gemeinsames Verhalten verwenden könnten.

**Wir sehen uns unsere Klassen an und erkennen, dass Wolf und Hund manche Verhalten gemeinsam haben könnten und dass Gleiches für Löwe, Tiger und Katze gilt.**

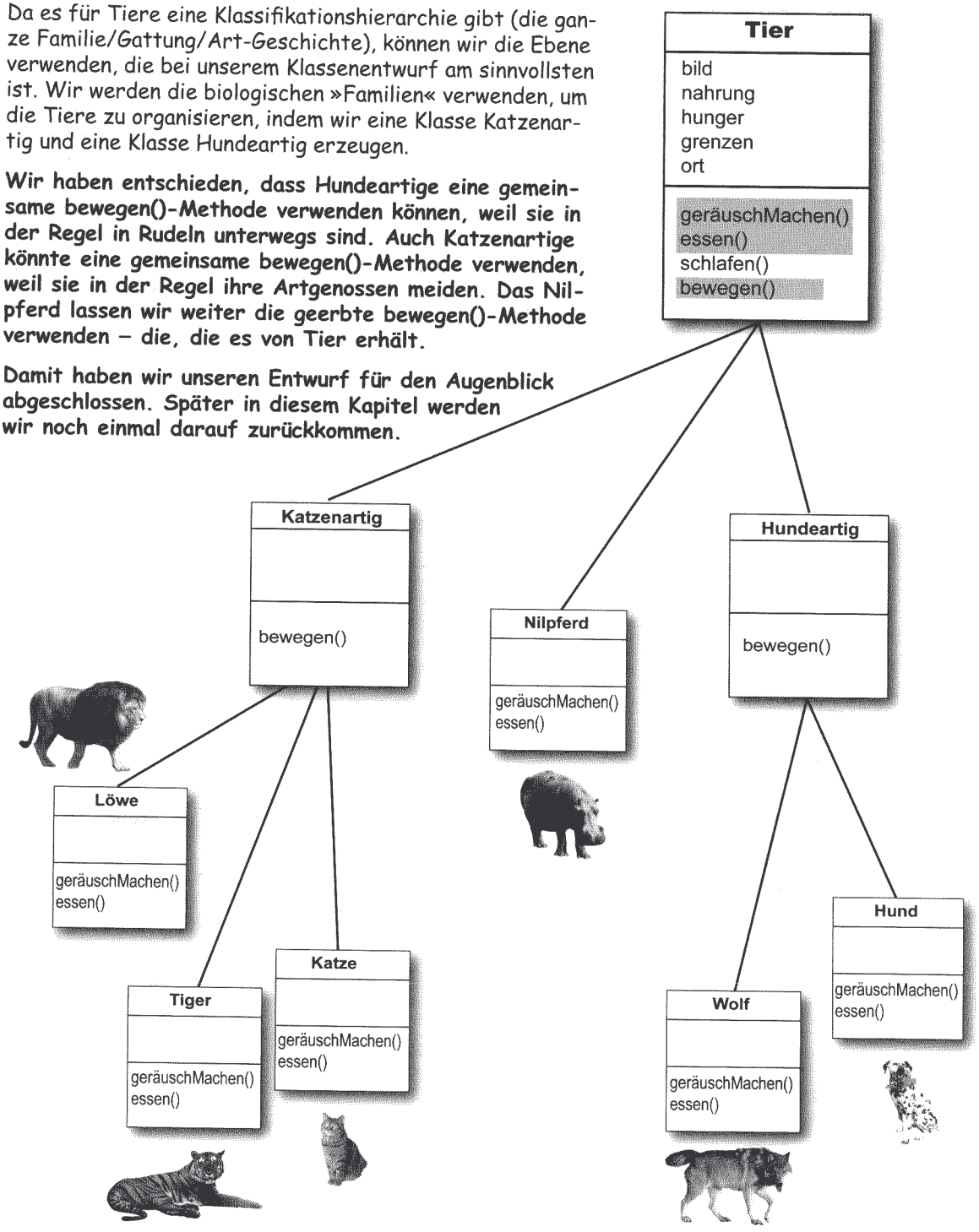


## 5 Die Klassenhierarchie abschließen

Da es für Tiere eine Klassifikationshierarchie gibt (die ganze Familie/Gattung/Art-Geschichte), können wir die Ebene verwenden, die bei unserem Klassenentwurf am sinnvollsten ist. Wir werden die biologischen »Familien« verwenden, um die Tiere zu organisieren, indem wir eine Klasse Katzenartig und eine Klasse Hundeartig erzeugen.

**Wir haben entschieden, dass Hundeartige eine gemeinsame bewegen()-Methode verwenden können, weil sie in der Regel in Rudeln unterwegs sind. Auch Katzenartige könnte eine gemeinsame bewegen()-Methode verwenden, weil sie in der Regel ihre Artgenossen meiden. Das Nilpferd lassen wir weiter die geerbte bewegen()-Methode verwenden – die, die es von Tier erhält.**

**Damit haben wir unseren Entwurf für den Augenblick abgeschlossen. Später in diesem Kapitel werden wir noch einmal darauf zurückkommen.**



# Welche Methode wird aufgerufen?

Die Klasse Wolf hat vier Methoden. Eine wird von Tier geerbt, eine von Hundeartige (die eigentlich eine überschriebene Version einer Methode in der Klasse Tier ist), und zwei werden in der Klasse Wolf überschrieben. Wenn Sie ein Wolf-Objekt erzeugen und es einer Referenzvariablen zuweisen, können Sie auf dieser Referenzvariablen den Punktoperator verwenden, um alle vier Methoden aufzurufen. Aber welche *Version* dieser Methoden wird dann aufgerufen?

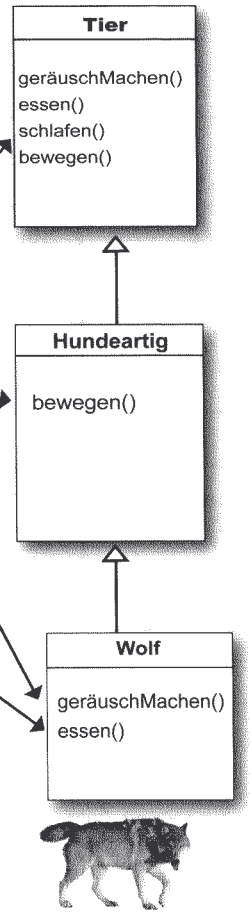
ein neues Wolf-Objekt erzeugen
```
Wolf w = new Wolf();
```

ruft die Version in Wolf auf
```
w.geräuschMachen();
```

ruft die Version in Hundeartig auf
```
w.bewegen();
```

ruft die Version in Wolf auf
```
w.essen();
```

ruft die Version in Tier auf
```
w.schlafen();
```




Wenn Sie eine Methode auf einer Objektreferenz aufrufen, rufen Sie immer die spezifischste Version dieser Methode für diesen Objekttyp auf.

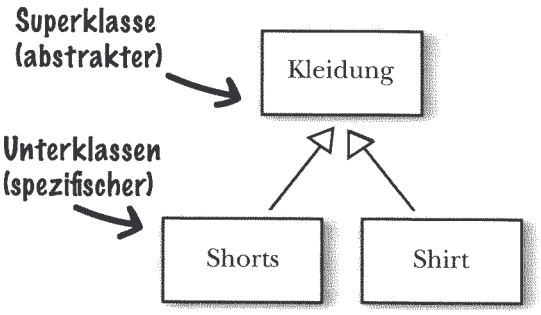
Mit anderen Worten: ***Die niedrigste Methode gewinnt!***

Die »niedrigste« meint dabei am niedrigsten im Vererbungsbaum. Hundeartig ist niedriger als Tier, und Wolf ist niedriger als Hundeartig. Wenn eine Methode auf einer Wolf-Referenzvariablen aufgerufen wird, bedeutet das also, dass die JVM erst in der Klasse Wolf sucht. Findet die JVM in der Klasse Wolf keine Version der Methode, arbeitet sie sich in der Hierarchie nach oben durch, bis sie eine passende Methode findet.

# Einen Vererbungsbaum entwerfen

| Klasse | Superklasse | Unterklassen |
|---|---|---|
| Kleidung | --- | Shorts, Shirt |
| Shorts | Kleidung | |
| Shirt | Kleidung | |

**Vererbungstabelle**




**Vererbungsklassendiagramm**

## Spitzen Sie Ihren Bleistift

Finden Sie sinnvolle Beziehungen. Füllen Sie die letzten beiden Spalten aus.

| Klasse | Superklasse | Unterklassen |
|---|---|---|
| Musiker | | |
| Rock-Star | | |
| Fan | | |
| Bassist | | |
| Konzertpianist | | |
| | | |

*Hinweis: Es kann nicht alles mit etwas anderem verbunden werden.*
*Hinweis: Sie dürfen die Klassen in der Liste ändern oder neue hinzufügen.*

Zeichnen Sie hier ein Vererbungsklassendiagramm.

## Es gibt keine Dummen Fragen

**F: Sie sagten, dass die JVM bei der Klasse beginnt, den Vererbungsbaum nach oben zu gehen, auf der man die Methode aufgerufen hat (wie Wolf in dem Beispiel auf der vorangegangenen Seite). Aber was passiert, wenn die JVM nirgendwo eine passende Methode findet?**

A: Gute Frage! Aber darüber müssen Sie sich nicht den Kopf zerbrechen. Der Compiler sichert, dass für einen spezifischen Referenztyp eine bestimmte Methode aufgerufen werden kann. Er sagt nur nicht (oder kümmert sich nicht darum), von welcher *Klasse* diese Methode zur Laufzeit kommt. Beim Wolf-Beispiel prüft der Compiler, ob eine schlafen()-Methode zugreifbar ist, aber es interessiert ihn nicht, dass schlafen() eigentlich in der Klasse Tier definiert (und von ihr geerbt) wird. Denken Sie daran, dass eine Klasse eine Methode *hat*, wenn sie die Methode *erbt*. *Wo* die geerbte Methode definiert ist (d.h. in welcher Superklasse sie definiert ist), macht für den Compiler keinen Unterschied. Aber zur Laufzeit **wählt die JVM immer die richtige**. Und die richtige heißt: ***die spezifischste Version für das jeweilige Objekt.***

# IS-A und HAS-A verwenden

Erinnern Sie sich daran, dass wir, wenn eine Klasse von der anderen erbt, davon sprechen, dass die Unterklasse die Superklasse *erweitert*? Wenn Sie wissen möchten, ob ein Ding ein anderes Ding erweitern soll, wenden Sie den IS-A-Test (Ist-Ein-Test) an.

Dreieck Ist-Eine Figur. Das funktioniert also.

Katze Ist-Ein katzenartiges Tier. Das funktioniert auch.

Chirurg Ist-Ein Arzt. Immer noch in Ordnung.

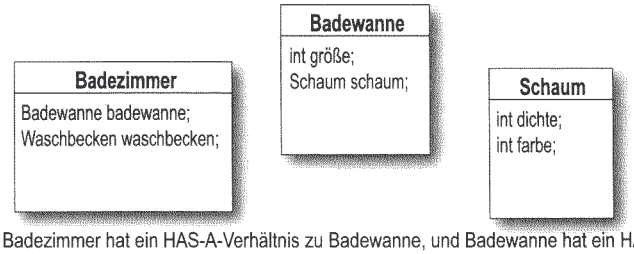
Badewanne erweitert Badezimmer – klingt vernünftig.

*Bis Sie darauf den IS-A-Test anwenden.*

Um zu prüfen, ob Sie Ihre Typen richtig entworfen haben, müssen Sie Folgendes fragen: »Kann man vernünftigerweise sagen, dass Typ X ein IS-A-Verhältnis mit Typ Y hat?« Hat er das nicht, wissen Sie, dass mit Ihrem Entwurf etwas nicht in Ordnung ist. Wenn wir den IS-A-Test anwenden, ist »Badewanne Ist-Ein Badezimmer« definitiv falsch.

Und was ist, wenn wir es zu »Badezimmer erweitert Badewanne« umkehren? Das funktioniert immer noch nicht. »Badezimmer Ist-Eine Badewanne« ist nicht richtig.

Badewanne und Badezimmer *stehen* in Beziehung, aber nicht über Vererbung. Badewanne und Badezimmer sind über eine HAS-A-Beziehung (Hat-Ein-Beziehung) verbunden. Kann man vernünftigerweise davon sprechen, dass »ein Badezimmer in einem HAS-A-Verhältnis zu einer Badewanne steht«? Wenn ja, heißt das, dass Badezimmer eine Badewanne-Instanzvariable hat. Mit anderen Worten: Badezimmer hat eine *Referenz* auf eine Badewanne, aber Badewanne *erweitert* Badezimmer nicht und umgekehrt.




Badezimmer hat ein HAS-A-Verhältnis zu Badewanne, und Badewanne hat ein HAS-A-Verhältnis zu Schaum. Niemand erbt von einem anderen (bzw. erweitert einen anderen).

# Warten Sie! Es kommt noch mehr!

Der IS-A-Test funktioniert *an jeder beliebigen Stelle* im Vererbungsbaum. Wenn Ihr Vererbungsbaum ordentlich entworfen ist, sollte der IS-A-Test sinnvoll sein, wenn Sie für eine *beliebige* Unterklasse fragen, ob sie ein IS-A-Verhältnis mit einem *beliebigen* ihrer Supertypen hat.

## Wenn Klasse B Klasse A erweitert, hat B ein IS-A-Verhältnis mit Klasse A.

## Das gilt an beliebiger Stelle im Vererbungsbaum. Wenn Klasse C Klasse B erweitert, besteht Klasse C den IS-A-Test für B *und* A.

Hundeartig erweitert Tier

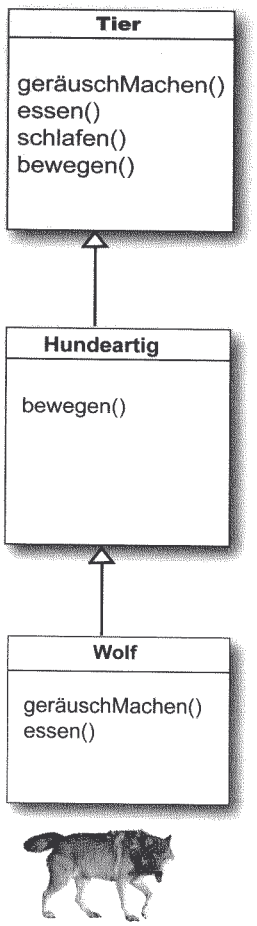
Wolf erweitert Hundeartig

Wolf erweitert Tier

Hundeartig Ist-Ein Tier

Wolf Ist-Ein Hundeartig

Wolf Ist-Ein Tier




Bei einem Vererbungsbaum wie dem, den wir hier zeigen, können Sie *immer* sagen, dass »**Wolf Tier erweitert**« oder »**Wolf ein IS-A-Verhältnis zu Tier hat**«. Es macht keinen Unterschied, ob Tier die Superklasse der Superklasse von Wolf ist. **So lange Tier sich in der Vererbungshierarchie *irgendwo* über Wolf befindet, ist »Wolf hat ein IS-A-Verhältnis mit Tier« immer wahr.**

Die Struktur des Tier-Vererbungsbaums sagt der Welt Folgendes:

»Wolf Ist-Ein Hundeartig, also kann Wolf alles tun, was ein Hundeartig tun kann. Und Wolf Ist-Ein Tier, also kann Wolf alles tun, was ein Tier tun kann.«

Es spielt keine Rolle, dass Wolf einige der Methoden in Tier oder Hundeartig überschreibt. Soweit die Welt (bzw. anderer Code) betroffen ist, kann ein Wolf diese vier Methoden ausführen. *Wie* er sie ausführt oder *in welcher Klasse sie überschrieben werden*, spielt keine Rolle. Ein Wolf kann geräuschMachen(), essen(), schlafen() und sich bewegen(), weil ein Wolf die Klasse Tier erweitert.

# Wie prüfen Sie, ob Ihre Vererbung stimmt?

Offensichtlich müssen Sie dabei mehr bedenken, als wir bisher behandelt haben. Aber im nächsten Kapitel (in dem wir bei Gelegenheit einige der Entwürfe, die wir in *diesem* Kapitel gemacht haben, verfeinern und verbessern werden) werden wir uns noch viel mehr OO-Fragen ansehen.

Im Augenblick ist es jedoch eine gute Richtschnur, einen IS-A-Test einzusetzen. Wenn »X Ist-Ein Y« sinnvoll ist, sollten beide Klassen (X und Y) wahrscheinlich im gleichen Vererbungsbaum leben. Dann ist die Wahrscheinlichkeit groß, dass sie die gleichen, einander überschneidenden Verhalten haben.

## Denken Sie allerdings daran, dass das Vererbungs-IS-A-Verhältnis nur in eine Richtung funktioniert!

»Dreieck Ist-Eine Figur« ist sinnvoll, also können Sie Dreieck Figur erweitern lassen.

Aber die Umkehrung – »Figur Ist-Ein Dreieck« – ist *nicht* sinnvoll. Also sollte Figur nicht Dreieck erweitern. Denken Sie auch daran, dass das IS-A-Verhältnis impliziert, dass X alles (und wahrscheinlich mehr) tun kann, was Y kann, wenn X Ist-Ein Y wahr ist.

### Spitzen Sie Ihren Bleistift

Setzen Sie ein Häkchen neben die Beziehungen, die sinnvoll sind.

- ☐ **Herd erweitert Küche**
- ☐ **Gitarre erweitert Instrument**
- ☐ **Person erweitert Angestellter**
- ☐ **Ferrari erweitert Auto**
- ☐ **Rührei erweitert Nahrung**
- ☐ **Beagle erweitert Haustier**
- ☐ **Gefäß erweitert Glas**
- ☐ **Metall erweitert Titan**
- ☐ **GratefulDead erweitert Band**
- ☐ **Blond erweitert Clever**
- ☐ **Getränk erweitert Martini**

*Hinweis: Wenden Sie einen IS-A-Test an.*

## Es gibt keine Dummen Fragen

**F: Wir verstehen, wie eine Unterklasse dazu kommt, eine Methode der Superklasse zu erben. Aber was ist, wenn die Superklasse die Unterklassenversion der Methode verwenden will?**

A: Eine Superklasse *weiß* normalerweise nicht einmal, dass sie Unterklassen hat. Sie könnten eine Klasse schreiben, und dann kommt viel, viel später jemand und erweitert sie. Aber selbst wenn der Autor einer Superklasse weiß, dass es eine Unterklassenversion einer Methode gibt (und diese verwenden möchte), gibt es so etwas wie eine *umgekehrte Vererbung* oder *Rückwärtsvererbung* nicht. Denken Sie darüber nach: Kinder erben von Eltern, nicht umgekehrt.

**F: Was mache ich in einer Unterklasse, wenn ich die Superklassenversion UND die Unterklassenversion einer Methode verwenden möchte? Anders gesagt: Ich möchte die Superklassenversion nicht vollständig *ersetzen*, ich möchte ihr nur etwas hinzufügen.**

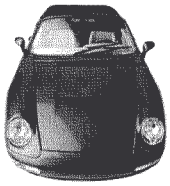
A: Das können Sie tun! Und das ist ein wichtiges Design-Feature. Übersetzen Sie das Wort »erweitern« mit »Ich möchte die Funktionalität der Superklasse *erweitern*«. Sie können die Methoden Ihrer Superklasse so entwerfen, dass sie Methodenimplementierungen enthalten, die für alle Unterklassen funktionieren, selbst wenn die Unterklassen gegebenenfalls noch mehr Code ›anhängen‹ müssen. In der überschreibenden Methode der Unterklasse können Sie die Superklassenversion aufrufen, indem Sie das Schlüsselwort **super** verwenden. Das ist, als sagten Sie: »Führ zuerst die Superklassenversion aus und komm dann zurück und arbeite meinen eigenen Code ab ...«

```
public void bewegen() {
    super.bewegen();
     // eigenes bewegen()-Zeug
}
```

Damit wird die geerbte Version von bewegen() aufgerufen, und anschließend kehrt man zurück, um den unterklassenspezifischen Code auszuführen.

# Wer kriegt den Porsche, wer das Porzellan?

## (oder: wie man erfährt, was eine Unterklasse von ihrer Superklasse erben kann)



Eine Unterklasse erbt die Member der Superklasse. Zu den Membern gehören die Instanzvariablen und die Methoden. Später im Buch werden wir uns aber noch andere geerbte Member ansehen. Eine Superklasse kann über die Zugriffsebene, die einem bestimmten Member verliehen wird, festlegen, welche Member Unterklassen erben können sollen.

Vom restriktivsten zum offensten Zugriff sind die vier Zugriffsebenen:

**private default protected public**

**Zugriffsebenen steuern,** ***wer was sieht,*** und sind für gut entworfenen, robusten Java-Code von entscheidender Bedeutung. Im Augenblick konzentrieren wir uns nur auf public und private. Die Regeln für diese beiden sind einfach:

**public (öffentliche) Member werden geerbt**
**private (private) Member werden nicht geerbt**

Wenn eine Unterklasse ein Member erbt, ist das, ***als hätte die Unterklasse das Member selbst definiert.*** Beim Figur-Beispiel hat Quadrat die Methoden `rotieren()` und `soundAbspielen()` geerbt, und für außen Stehende (anderen Code) sieht es so aus, dass die Klasse Quadrat einfach eine `rotieren()` – und eine `soundAbspielen()`-Methode *hat.*

Zu den Membern einer Klasse gehören die Variablen und Methoden, die in der Klasse definiert werden, und alles, was von einer Superklasse geerbt wird.

Hinweis: Mehr Informationen zu default und protected erhalten Sie in Kapitel 17 (Deployment) und Anhang B.

# Vererbung kann man gut gebrauchen – man kann sie aber auch missbrauchen!

Auch wenn einige der Gründe hinter diesen Regeln erst später in diesem Buch klar werden, hilft es Ihnen jetzt schon, bessere Vererbungsentwürfe aufzubauen, wenn Sie einige dieser Regeln *kennen*.

**JA, verwenden** Sie Vererbung, wenn eine Klasse ein spezifischerer Typ einer Superklasse ist. Beispiel: Weide *ist ein* spezifischerer Typ von Baum, also ist es sinnvoll, dass Weide Baum *erweitert*.

**JA, denken** Sie über Vererbung nach, wenn Sie Verhalten (implementierten Code) haben, der von mehreren Klassen des gleichen allgemeinen Typs gemeinsam verwendet wird. Beispiel: Quadrat, Kreis und Dreieck müssen alle rotieren und Sound abspielen. Also könnte es sinnvoll sein, diese Funktionalität in eine Superklasse Figur zu stecken und so für eine vereinfachte Wartung und Erweiterbarkeit zu sorgen. Achten Sie jedoch darauf, dass Vererbung zwar eins der Schlüssel-Features der objektorientierten Programmierung, aber trotzdem nicht notwendigerweise das beste Mittel ist, Wiederverwendbarkeit von Verhalten zu erreichen. Es bringt Sie auf den Weg und ist oft die richtige Entwurfsentscheidung, aber Entwurfsmuster helfen Ihnen, andere subtilere und flexiblere Optionen wahrzunehmen. Wenn Sie Entwurfsmuster noch nicht kennen, empfehlen wir Ihnen als Ergänzung zu diesem Buch die Lektüre von *Entwurfsmuster von Kopf bis Fuß*!

**NEIN, verwenden** Sie Vererbung **NICHT** einfach nur, um Code aus einer anderen Klasse benutzen zu können, wenn das Verhältnis zwischen der Superklasse und der Unterklasse eine der beiden oben angegebenen Regeln verletzt. Stellen Sie sich beispielsweise vor, Sie hätten in der Wecker-Klasse speziellen Druck-Code geschrieben und bräuchten jetzt Druck-Code in der Klasse Klavier und ließen deswegen Klavier Wecker erweitern. Das ergibt keinen Sinn! Ein Klavier ist *kein* spezifischerer Typ von Wecker. (Also sollte sich der Druck-Code in einer Klasse Drucker befinden, die alle druckbaren Objekte über ein HAS-A-Verhältnis nutzen können.)

**NEIN, verwenden** Sie Vererbung **NICHT,** wenn die Superklasse und die Unterklasse den IS-A-Test nicht bestehen. Fragen Sie sich immer, ob die Unterklasse ein spezifischerer Typ der Superklasse ist. Beispiel: »Tee Ist-Ein Getränk« ist sinnvoll. »Getränk Ist-Ein Tee« ist nicht sinnvoll.

## Punkt für Punkt

- Eine Unterklasse *erweitert* eine Superklasse.
- Eine Unterklasse erbt alle *public* (öffentlichen) Instanzvariablen und Methoden der Superklasse, aber nicht die *private* (privaten) Instanzvariablen und Methoden der Superklasse.
- Geerbte Methoden *können* überschrieben werden; Instanzvariablen *können nicht* überschrieben werden (auch wenn sie in der Unterklasse *neu definiert* werden können, aber das ist nicht das Gleiche, und in der Regel ist es eigentlich auch fast nie erforderlich).
- Verwenden Sie einen IS-A-Test, um zu prüfen, ob Ihre Vererbungshierarchie gültig ist. Wenn X Y *erweitert*, muss X *Ist-Ein* Y sinnvoll sein.
- Das IS-A-Verhältnis funktioniert nur in eine Richtung. Ein Nilpferd ist ein Tier, aber nicht alle Tiere sind Nilpferde.
- Wenn eine Methode in einer Unterklasse überschrieben und diese Methode auf einer Instanz der Unterklasse aufgerufen wird, wird die überschriebene Version der Methode aufgerufen. (*Die niedrigste gewinnt.*)
- Wenn Klasse B A erweitert und C B erweitert, hat B ein IS-A-Verhältnis mit A und C ein IS-A-Verhältnis mit B und C auch ein IS-A-Verhältnis mit A.

# Was also bringt Ihnen der ganze Vererbungskram?

Aus OO-Sicht sind Sie einen großen Schritt weiter, wenn Sie mit Vererbung arbeiten. Sie werden doppelten Code los, indem Sie allgemeines Verhalten aus einer Gruppe von Klassen herausziehen und es in eine Superklasse stecken. Auf diese Weise müssen Sie nur eine Stelle aktualisieren, wenn Sie Ihren Code überarbeiten müssen, *und diese Änderungen werden wie von Zauberhand von allen Klassen wiedergegeben, die das Verhalten ändern.* Eigentlich hat das nichts mit Magie zu tun, aber es *ist* ziemlich einfach, die Änderungen vorzunehmen und die Klasse neu zu kompilieren. Das war's dann schon. **Sie müssen die Unterklassen nicht anrühren!**

**Liefern Sie einfach die gerade geänderte Superklasse aus, und schon nutzen alle Klassen, die sie erweitern, automatisch die neue Version.**

Ein Java-Programm ist nichts anderes als ein Haufen Klassen. Die Unterklassen müssen also nicht neu kompiliert werden, damit sie die neue Version der Superklasse verwenden. Solange die Superklasse für die Unterklassen nichts *zerbricht*, ist alles in Ordnung. (Was das Wort »zerbrechen« in diesem Zusammenhang bedeutet, werden wir später in diesem Buch behandeln. Im Augenblick können Sie sich das so denken, dass die Superklasse etwas verändert, von dem die Unterklasse abhängt, wie ein Argument einer bestimmten Methode, einen Rückgabewert oder einen Methodennamen usw.)

### 1 Sie vermeiden doppelten Code.

Stecken Sie allgemeinen Code an einen Ort und lassen Sie die Unterklassen diesen Code von der Superklasse erben. Wenn Sie dieses Verhalten ändern möchten, müssen Sie es nur an diesem einen Ort ändern, und alle anderen (alle Unterklassen) sehen die Änderung.

### 2 Sie definieren ein allgemeines Protokoll für eine Gruppe von Klassen.

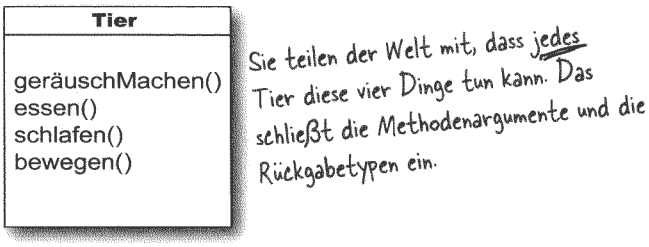
# Mit Vererbung können Sie garantieren, dass alle Klassen, die unter einem bestimmten Supertyp gruppiert werden, alle Methoden haben, die der Supertyp hat.*

## Anders gesagt: Sie definieren über Vererbung ein gemeinsames Protokoll für eine Gruppe von Klassen.

Wenn Sie in einer Superklasse Methoden definieren, die von Unterklassen geerbt werden können, erklären Sie für anderen Code eine Art Protokoll, das sagt: »Alle meine Untertypen (d.h. Unterklassen) können diese Dinge tun, sie bieten dazu Methoden, die so aussehen ...«

Man könnte also sagen, dass Sie einen *Vertrag* entwerfen.

Die Klasse Tier definiert ein gemeinsames Protokoll für alle Untertypen von Tier:




Und denken Sie daran, wenn wir *irgendein Tier* sagen, meinen wir Tier *und jede Klasse, die Tier erweitert.* Das heißt wiederum, *jede Klasse, die Tier in der Vererbungshierarchie irgendwo über sich hat.*

Aber zum richtig coolen Teil sind wir noch gar nicht gekommen, das Beste – *Polymorphie* – haben wir uns für das Ende aufgehoben.

Wenn Sie einen Supertyp für eine Gruppe von Klassen definieren, *kann jede Unterklasse dieses Supertyps an jeder Stelle eingesetzt werden, an der der Supertyp erwartet wird.*

Wie bitte?

Keine Sorge, wir haben mit der Erklärung noch nicht einmal angefangen. In zwei Seiten werden Sie Experte sein.

*Wenn wir »alle Methoden« sagen, meinen wir »alle *vererbbaren* Methoden«, was im Augenblick eigentlich »alle *public* Methoden« heißt, auch wenn wir diese Definition später noch etwas verfeinern werden.

## Ich mache mir Gedanken ...

Weil Sie lernen, die Vorteile der Polymorphie auszunutzen.

## Das ist wichtig für mich ...

Weil Sie dazu übergehen, mit einer Referenz, die als Referenz für den Supertyp deklariert ist, Objekte der Unterklasse zu referenzieren.

## Und das bedeutet für mich ...

Dass Sie lernen, richtig flexiblen Code zu schreiben. Code, der sauberer (effizienter, einfacher) ist. Code, der nicht nur leichter zu *entwickeln*, sondern auch viel, viel leichter zu *erweitern* ist, und das auf eine Weise, an die Sie noch nicht einmal gedacht haben, als Sie den Code geschrieben haben.

Das heißt, dass Sie Urlaub in der Karibik machen können, während Ihre Kollegen das Programm aktualisieren und dazu Ihren Quellcode wahrscheinlich gar nicht brauchen.

Wie es funktioniert, sehen Sie auf der nächsten Seite.

Wir wissen nicht, wie das bei Ihnen ist, aber wir finden die Sache mit dem Urlaub in der Karibik besonders motivierend.

**Um Ihnen zu zeigen, wie Polymorphie funktioniert, müssen wir einen Schritt zurückgehen und uns noch einmal ansehen, wie wir *normalerweise* eine Referenz deklarieren und ein Objekt erzeugen ...**

## Die 3 Schritte der Objekt-deklaration und -zuweisung

```
1          3   2
Hund meinHund = new Hund();
```

### 1 Eine Referenzvariable deklarieren

```
Hund meinHund = new Hund();
```

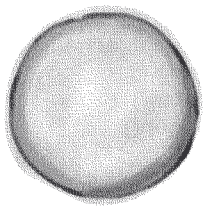
Teilt der JVM mit, Speicherplatz für eine Referenzvariable zu allozieren. Die Referenzvariable hat auf ewig den Typ Hund. Eine Fernsteuerung also, die Knöpfe zur Steuerung eines Hundes, aber keine Knöpfe zur Steuerung einer Katze, eines Buttons oder eines Sockets hat.

Hund

### 2 Ein Objekt erzeugen

```
Hund meinHund = new Hund();
```

Teilt der JVM mit, auf dem Garbage Collectible Heap Speicherplatz für ein neues Hund-Objekt zu allozieren.

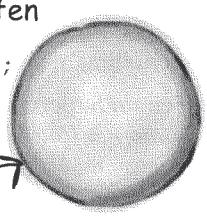


Hund-Objekt

### 3 Das Objekt mit der Referenz verknüpfen

```
Hund meinHund = new Hund();
```

Weist der Referenzvariablen meinHund das neue Hund-Objekt zu. Anders gesagt, ***programmiert die Fernsteuerung.***

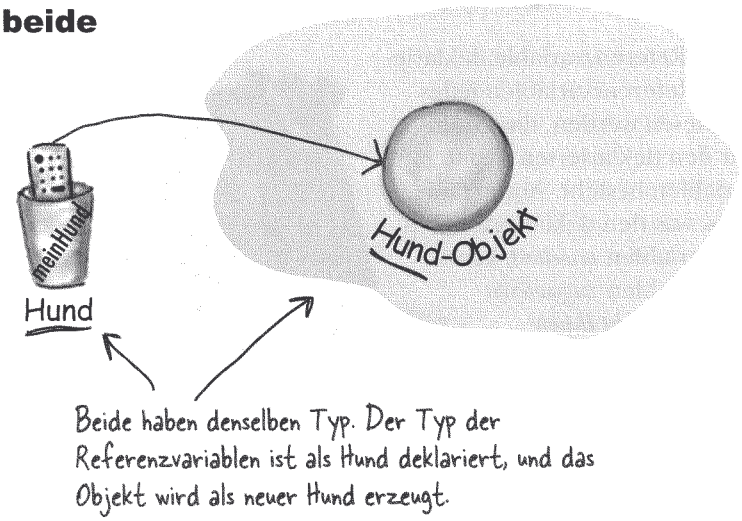


Hund-Objekt

Hund

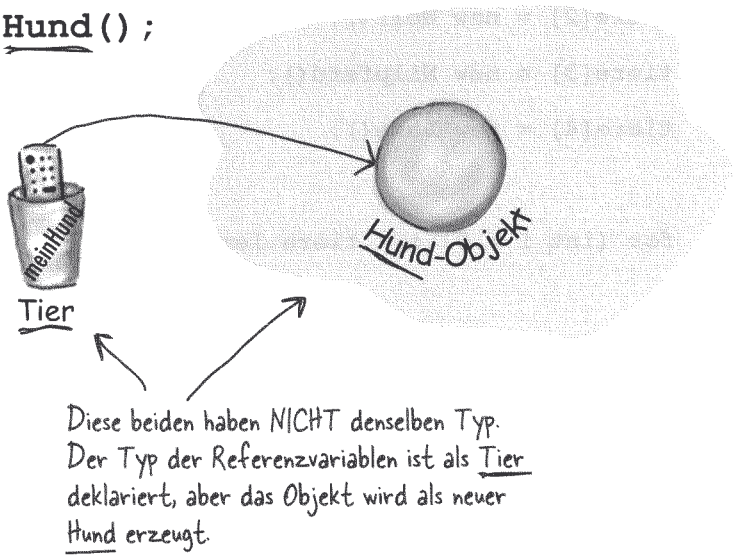
**Der entscheidende Punkt ist, dass Referenztyp UND Objekttyp gleich sind.**

**In diesem Fall sind sie beide Hund.**




**Aber mit Polymorphie können Referenz und Objekt *unterschiedlich* sein.**

```
Tier meinHund = new Hund();
```



## Mit Polymorphie kann der Typ der Referenz eine Superklasse des tatsächlichen Objekttyps sein.

Wenn Sie eine Referenzvariable deklarieren, kann der Referenzvariablen jedes Objekt zugewiesen werden, das einen IS-A-Test für den deklarierten Typ der Referenzvariablen besteht. Anders formuliert: Alles, was den deklarierten Typ der Referenzvariablen *erweitert*, kann der Referenzvariablen *zugewiesen* werden. ***Das ermöglicht Ihnen beispielsweise, Dinge wie polymorphe Arrays zu erstellen.***

## Gut. Gut. Vielleicht hilft ein Beispiel.

```
Tier[] tiere = new Tier[5];

tiere[0] = new Hund();
tiere[1] = new Katze();
tiere[2] = new Wolf();
tiere[3] = new Nilpferd();
tiere[4] = new Löwe();

for (int i = 0; i < tiere.length; i++) {

   tiere[i].essen();

   tiere[i].bewegen();
}
```

Ein Array des Typs Tier deklarieren, d.h. ein Array, das Objekte vom Typ Tier speichert.

Aber sehen Sie, was Ihnen das ermöglicht ... Sie können JEDE BELIEBIGE Unterklasse von Tier in das Tier-Array stecken!

Und hier ist der beste polymorphe Teil (die Legitimation des ganzen Beispiels): Sie können das Array in einer Schleife durchlaufen und eine der Methoden der Klasse Tier aufrufen, und jedes Objekt macht genau, was es machen soll!

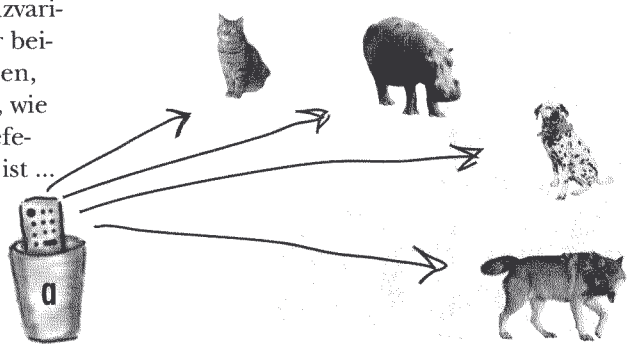
Wenn »i« gleich 0 ist, befindet sich ein Hund am Index 0 des Arrays, also erhalten Sie die essen()-Methode von Hund. Wenn »i« gleich 1 ist, erhalten Sie die essen()-Methode von Katze.

Und genau so ist es auch bei bewegen().

## Warten Sie! Da kommt noch mehr!

### Sie können auch polymorphe Argumente und Rückgabetypen haben.

Nehmen wir an, dass Sie eine Referenzvariable eines Supertyps deklarieren, Tier beispielsweise, und ihr ein Objekt zuweisen, vielleicht einen Hund. Überlegen Sie, wie es funktionieren könnte, wenn die Referenz ein Argument für eine Methode ist ...



```
class Tierarzt {
  public void spritzeGeben(Tier a) {
    // schreckliche Dinge mit dem Tier
    // am anderen Ende des Parameters 'a' tun
    a.geräuschMachen();
  }
}
```

Der Parameter »a« kann JEDEN BELIEBIGEN Tier-Typ als Argument nehmen. Wenn der Tierarzt die Spritze verabreicht hat, sagt er dem Tier, dass es geräuschMachen() soll, und egal welches Tier sich tatsächlich auf dem Heap befindet, seine geräuschMachen()-Methode wird ausgeführt.

```
class Tierbesitzer {
  public void start() {
    Tierarzt t = new Tierarzt();
    Hund h = new Hund();
    Nilpferd n = new Nilpferd();
    t.spritzeGeben(h);
    t.spritzeGeben(n);
  }
}
```

Die spritzeGeben()-Methode von Tierarzt kann jedes Tier verarzten, das Sie ihr überreichen. Das funktioniert, wenn das Objekt, das Sie als Argument hereinreichen, eine Unterklasse von Tier ist.

geräuschMachen() von Hund wird ausgeführt.

geräuschMachen() von Nilpferd wird ausgeführt.

## Mit Polymorphie können Sie Code schreiben, der nicht geändert werden muss, wenn Sie einem Programm eine neue Unterklasse hinzufügen.

Denken Sie an die Klasse Tierarzt? Wenn Sie beim Schreiben der Klasse Tierarzt Argumente verwenden, deren deklarierter Typ *Tier* ist, kann Ihr Code mit allen Tier-*Unterklassen* umgehen. Möchten andere Ihre Tierarzt-Klasse verwenden, heißt das, dass sie nur sicherstellen müssen, dass *ihre* neuen Tier-Typen die Klasse Tier erweitern. Die Tierarzt-Methoden funktionieren dann immer noch, selbst wenn die Klasse Tierarzt geschrieben wurde, ohne dass die neuen Tier-Untertypen, mit denen der Tierarzt arbeitet, bekannt waren.

## KOPF-NUSS

Was garantiert, dass Polymorphie auf diese Weise funktioniert? Warum kann man immer sicher davon ausgehen, dass jede der *Unterklassen* die erwarteten Methoden hat, wenn Sie sie auf der *Superklasse* (mit dem Punktoperator auf der Referenz vom Typ der Superklasse) aufrufen?

## Es gibt keine Dummen Fragen

**F: Gibt es praktische Grenzen für die Anzahl von Ebenen von Unterklassen? Wie tief kann man gehen?**

A: Wenn Sie sich die Java-API ansehen, werden Sie sehen, dass die meisten Vererbungshierarchien breit, aber nicht tief sind. Die meisten sind nicht mehr als ein oder zwei Ebenen tief. Es gibt allerdings auch Ausnahmen (insbesondere bei den GUI-Klassen). Sie werden feststellen, dass es in der Regel sinnvoller ist, Vererbungsbäume flach zu halten, aber eine festgeschriebene Grenze gibt es nicht (gut, keine, auf die Sie jemals stoßen werden).

**F: Äh. Ich habe mir gerade was überlegt ... wenn man keinen Zugriff auf den Quellcode einer Klasse hat, aber ändern möchte, wie eine Methode dieser Klasse funktioniert, könnte man dann nicht eine Unterklasse verwenden, um das zu tun? Um die »schlechte« Klasse zu erweitern und die Methode mit dem eigenen besseren Code zu überschreiben?**

A: Ja. Das ist eins der coolen Features von OO, und manchmal können Sie dadurch vermeiden, die Klasse neu zu schreiben oder den Programmierer aufzuspüren, der den Quellcode versteckt hat.

**F: Kann man *jede beliebige* Klasse erweitern? Oder ist das wie bei den Klassen-Membern, bei denen man Klassen, die private sind, nicht erweitern kann ...**

A: So was wie private Klassen gibt es nicht, außer in einem sehr speziellen Fall, den man als *innere* Klasse bezeichnet, den wir uns bisher aber noch nicht angesehen haben. Es *gibt jedoch* drei Dinge, die verhindern können, dass von einer Klasse Unterklassen abgeleitet werden.

Das Erste ist die Zugriffskontrolle. Auch wenn eine Klasse *nicht* als `private` markiert werden *kann*, kann sie nicht-öffentlich sein (das, was Sie erhalten, wenn Sie eine Klasse nicht explizit als `public` deklarieren). Von einer nicht-öffentlichen Klasse können nur innerhalb des gleichen Packages Unterklassen abgeleitet werden. Klassen in einem anderen Package können keine Unterklassen der nicht-öffentlichen Klasse sein (oder sie auch nur *verwenden*, falls Sie das interessiert).

Das Zweite, das verhindert, dass von einer Klasse Unterklassen gebildet werden, ist das Modifizierer-Schlüsselwort `final`. Wenn eine Klasse als final markiert ist, heißt das, dass sie das Ende einer Vererbungslinie ist. Niemand kann irgendwann eine finale Klasse erweitern.

Das Dritte ist, dass von einer Klasse keine Unterklassen gebildet werden, wenn sie nur `private` Konstruktoren hat (Konstruktoren werden wir uns in Kapitel 9 ansehen).

**F: Warum sollte man eine finale Klasse machen wollen? Welchen Vorteil hat es, wenn man verhindert, dass von einer Klasse Unterklassen gebildet werden?**

A: Normalerweise macht man seine Klassen nicht final. Aber wenn man Sicherheit braucht – die Sicherheit, dass die Methoden immer so funktionieren, wie man sie geschrieben hat (weil sie nicht überschrieben werden können)–, dann kann eine finale Klasse diese bieten. Aus diesem Grund sind viele Klassen in der Java-API final. Die Klasse String ist beispielsweise final, weil ... na stellen Sie sich nur das Chaos vor, das eintreten könnte, wenn jemand vorbeikommt und einfach das Verhalten von Strings ändert!

**F: Kann man eine *Methode* final machen, ohne die ganze *Klasse* final zu machen?**

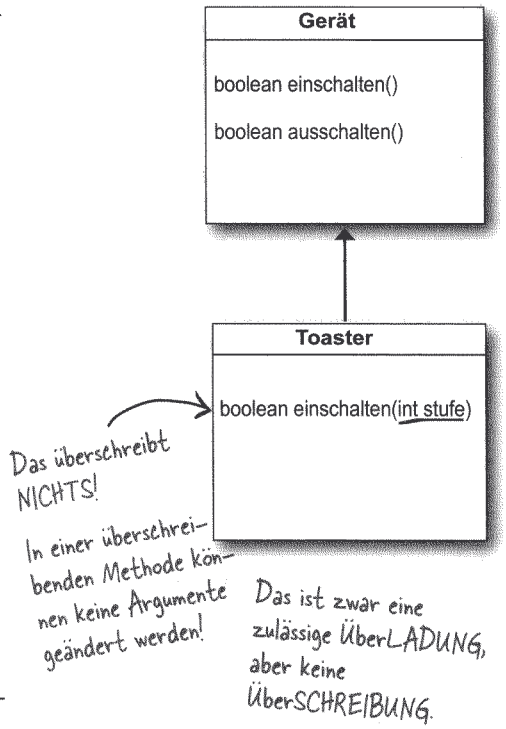
A: Wenn Sie eine bestimmte Methode davor schützen möchten, überschrieben zu werden, können Sie diese *Methode* mit dem Modifier `final` markieren. Markieren Sie die ganze *Klasse* als final, wenn Sie sichern möchten, dass *keine* der Methoden in der Klasse jemals überschrieben wird.

# Den Vertrag einhalten: Regeln zum Überschreiben

Wenn Sie eine Methode einer Superklasse überschreiben, bestätigen Sie, dass Sie den Vertrag einhalten werden. Den Vertrag beispielsweise, der sagt: »Ich erwarte keine Argumente und liefere einen Booleschen Wert zurück.« Mit anderen Worten, die Argumente und Rückgabetypen Ihrer überschreibenden Methode müssen für die Welt *genau* so aussehen wie die überschriebene Methode in der Superklasse.

**Die Methoden *sind* der Vertrag.**

Wenn Polymorphie funktioniert, muss die Toaster-Version der überschriebenen Methode von Gerät zur Laufzeit funktionieren. Denken Sie daran, dass der Compiler auf den Typ der Referenz blickt, um zu entscheiden, ob Sie eine bestimmte Methode auf dieser Referenz aufrufen können. Bei einer Gerät-Referenz auf einen Toaster kümmert sich der Compiler nur darum, ob die Klasse *Gerät* die Methode besitzt, die Sie auf der Gerät-Referenz aufrufen. Aber zur Laufzeit blickt die JVM nicht auf den Typ der *Referenz* (Gerät), sondern auf das eigentliche *Toaster*-Objekt auf dem Heap. Wenn der Compiler den Methodenaufruf bereits *zugelassen* hat, kann er also nur funktionieren, wenn die überschreibende Methode die gleichen Argumente und Rückgabetypen hat. Andernfalls ruft jemand mit einer Gerät-Referenz einschalten() als argumentlose Methode auf, obwohl es in Toaster eine Version gibt, die ein int erwartet. Welche wird zur Laufzeit aufgerufen? Die in Gerät. Anders gesagt: ***Die einschalten(int stufe)-Methode in Toaster überschreibt nichts!***



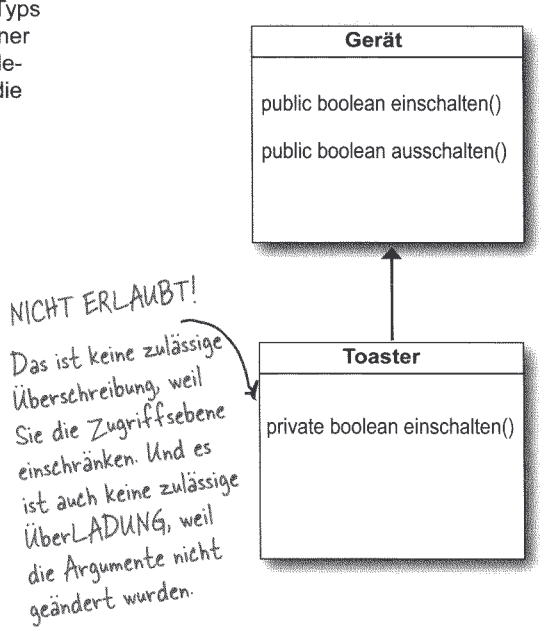

## 1 Die Argumente müssen gleich und die Rückgabetypen müssen kompatibel sein.

Der Vertrag der Superklasse definiert, wie anderer Code eine Methode verwenden kann. Was die Superklasse als Argument erwartet, muss auch die Unterklasse, die die Methode überschreibt, als Argument verwenden. Und egal was die Superklasse als Rückgabetyp deklariert, die überschreibende Methode muss den gleichen Rückgabetyp oder einen Untertyp dieses Typs deklarieren. Denken Sie daran, dass garantiert wird, dass ein Objekt einer Unterklasse dazu in der Lage ist, alles zu tun, was seine Superklasse deklariert. Deswegen ist es sicher, einen Untertyp zurückzuliefern, wenn die Superklasse erwartet wird.

## 2 Die Zugreifbarkeit der Methode kann nicht geringer sein.

Das heißt, dass die Zugriffsebene gleich oder großzügiger sein muss. Das bedeutet beispielsweise, dass Sie keine öffentliche Methode überschreiben und dabei privat machen können. Was für ein Schock wäre das für den Code, der versucht, das aufzurufen, was er (zur Kompilierzeit) für eine öffentliche Methode hält, wenn plötzlich die JVM zur Laufzeit die Tür zuschlägt, weil die zur Laufzeit aufgerufene überschreibende Version privat ist!

Bisher haben wir zwei Zugriffsebenen kennen gelernt: private (privat) und public (öffentlich). Die anderen beiden finden Sie in Anhang B. Es gibt auch noch eine weitere Regel beim Überschreiben, die mit dem Exception-Handling zu tun hat. Aber wir werden bis zum Kapitel über die Exceptions warten, bis wir das behandeln.



# Eine Methode überladen

Methoden zu überladen heißt nichts anderes, als dass man zwei Methoden mit dem gleichen Namen, aber unterschiedlichen Argumentlisten hat. Punkt. Überladene Methoden haben nichts mit Polymorphie zu tun.

Durch Überladen können Sie zur Bequemlichkeit Ihrer Benutzer mehrere Versionen einer Methode mit unterschiedlichen Argumentlisten bereitstellen. Nehmen Sie beispielsweise an, Sie haben eine Methode, die nur ein int erwartet, und aufrufender Code muss beispielsweise ein double in ein int umwandeln, bevor er Ihre Methode aufrufen kann. Wenn Sie die Methode dann mit einer anderen Version überladen, die ein double erwartet, würden Sie dem Aufrufer die Sache erleichtern. Mehr davon werden Sie sehen, wenn wir uns im Objekt-Lebensdauer-Kapitel 9 Konstruktoren ansehen.

Da eine überladende Methode nicht versucht, den Polymorphie-Vertrag zu erfüllen, der von der Superklasse definiert wird, besitzen überladene Methoden eine erheblich größere Flexibilität.

### 1 Die Rückgabetypen können anders sein.

Bei überladenen Methoden können Sie die Rückgabetypen beliebig ändern, solange auch die Argumentlisten verschieden sind.

### 2 Sie dürfen nicht NUR den Rückgabetyp ändern.

Wenn nur der Rückgabetyp anders ist, ist das keine gültige Über*ladung* – der Compiler geht dann davon aus, dass Sie versuchen, die Methode zu über*schreiben*. Und auch *das* ist nicht gültig, es sei denn, der Rückgabetyp ist ein Untertyp des Rückgabetyps, der in der Superklasse deklariert wird. Um eine Methode zu überladen, MÜSSEN Sie die Argumentliste ändern. Es steht Ihnen frei, dabei *auch* den Rückgabetyp in etwas beliebiges anderes zu ändern.

### 3 Sie *können* die Zugriffsebenen in beliebige Richtung ändern.

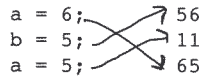
Es steht Ihnen frei, eine Methode mit einer Methode zu überladen, die restriktiver ist. Es spielt keine Rolle, da die neue Methode nicht verpflichtet ist, den Vertrag der überladenen Methode zu erfüllen.

Eine überladene Methode ist einfach eine andere Methode, die zufällig den gleichen Methodennamen hat. Das hat nichts mit Polymorphie und Vererbung zu tun. Eine überladene Methode ist NICHT das Gleiche wie eine überschriebene Methode.

## Gültige Beispiele für das Überladen von Methoden:

```
public class Überladen {

  String eindeutigeID;

  public int zahlenHinzufügen(int a, int b) {
    return a + b;
  }

  public double zahlenHinzufügen(double a,
                                 double b) {
    return a + b;
  }

  public void setEindeutigeID(String dieID) {
    // massenhaft Validierungscode und dann:
    eindeutigeID = dieID;
  }

  public void setEindeutigeID(int zahlID) {
    String zahlString = "" + zahlID;
    setEindeutigeID(zahlString);
  }
}
```

# Vermischte Nachrichten




Unten wird ein kurzes Java-Programm aufgeführt. Ein Block des Programms fehlt! Ihre Aufgabe ist es, mögliche Codeblöcke (auf der linken Seite) den Ausgaben zuzuordnen, die Sie sehen würden, wenn der entsprechende Block eingefügt wird. Es werden nicht alle Zeilen mit Ausgabe benötigt, und manche der Ausgaben können mehrfach verwendet werden. Ziehen Sie Linien, die die Codekandidaten mit den passenden Kommandozeilen-Ausgaben verbinden.

## Das Programm:

```
class A {
  int ivar = 7;
  void m1() {
    System.out.print("As m1, ");
  }
  void m2() {
    System.out.print("As m2, ");
  }
  void m3() {
    System.out.print("As m3, ");
  }
}

class B extends A {
  void m1() {
    System.out.print("Bs m1, ");
  }
}

class C extends B {
  void m3() {
    System.out.print("Cs m3, "+(ivar + 6));
  }
}

public class Vermischt2 {
    public static void main(String [] args) {
      A a = new A();
      B b = new B();
      C c = new C();
      A a2 = new C();
      [                    ]
    }
}
```

Hier kommen die Codekandidaten rein (drei Zeilen).

## Codekandidaten:

```
b.m1();
c.m2();
a.m3();
```

---

```
c.m1();
c.m2();
c.m3();
```

---

```
a.m1();
b.m2();
c.m3();
```

---

```
a2.m1();
a2.m2();
a2.m3();
```

## Ausgabe:

```
As m1, As m2, Cs m3, 6

Bs m1, As m2, As m3,

As m1, Bs m2, As m3,

Bs m1, As m2, Cs m3, 13

Bs m1, Cs m2, As m3,

Bs m1, As m2, Cs m3, 6

As m1, As m2, Cs m3, 13
```

# Spielen Sie Compiler

Welches der A/B-Methodenpaare, die auf der rechten Seite aufgeführt werden, lässt sich kompilieren und produziert die angezeigte Ausgabe, wenn es in die Klassen auf der linken Seite eingefügt wird? (Die Methode A wird in die Klasse Monster und die Methode B wird in die Klasse Vampir eingefügt.)

```
public class MonsterTestlauf {
  public static void main(String [] args) {
    Monster [] monster = new Monster[3];
    monster[0] = new Vampir();
    monster[1] = new Drache();
    monster[2] = new Monster();
    for(int x = 0; x < 3; x++) {
      monster[x].erschrecken(x);
    }
  }
}

class Monster {
  A
}

class Vampir extends Monster {
  B
}

class Drache extends Monster {
  boolean erschrecken(int grad) {
    System.out.println("Feuerhauch");
    return true;
  }
}
```

```
Datei Bearbeiten Fenster Hilfe RetteSichWerKann

% java MonsterTestLauf
Beißen, Beißen
Feuerhauch
Huh
```

1

A
```
boolean erschrecken(int d) {
  System.out.println("Huh");
  return true;
}
```

B
```
boolean erschrecken(int x) {
  System.out.println("Beißen, Beißen");
  return false;
}
```

2

A
```
boolean erschrecken(int x) {
  System.out.println("Huh");
  return true;
}
```

B
```
int erschrecken(int f) {
  System.out.println("Beißen, Beißen");
  return 1;
}
```

3

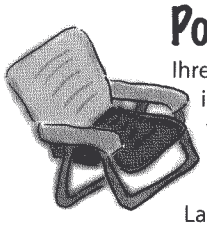
A
```
boolean erschrecken(int x) {
  System.out.println("Huh");
  return false;
}
```

B
```
boolean verängstigen(int x) {
  System.out.println("Beißen, Beißen");
  return true;
}
```

4

A
```
boolean erschrecken(int z) {
  System.out.println("Huh");
  return true;
}
```

B
```
boolean erschrecken(byte b) {
  System.out.println("Beißen, Beißen");
  return true;
}
```



# Pool-Puzzle

Ihre ***Aufgabe*** ist es, die Codeschnipsel aus dem Pool zu nehmen und sie in die Lücken im Code einzusetzen. Jedes Schnipsel darf **mehrmals** verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Ihr ***Ziel*** ist es, einen Satz von Klassen zu erstellen, die sich kompilieren und ausführen lassen und die hier gezeigte Ausgabe erzeugen. Lassen Sie sich nicht täuschen – das ist schwerer, als es aussieht.

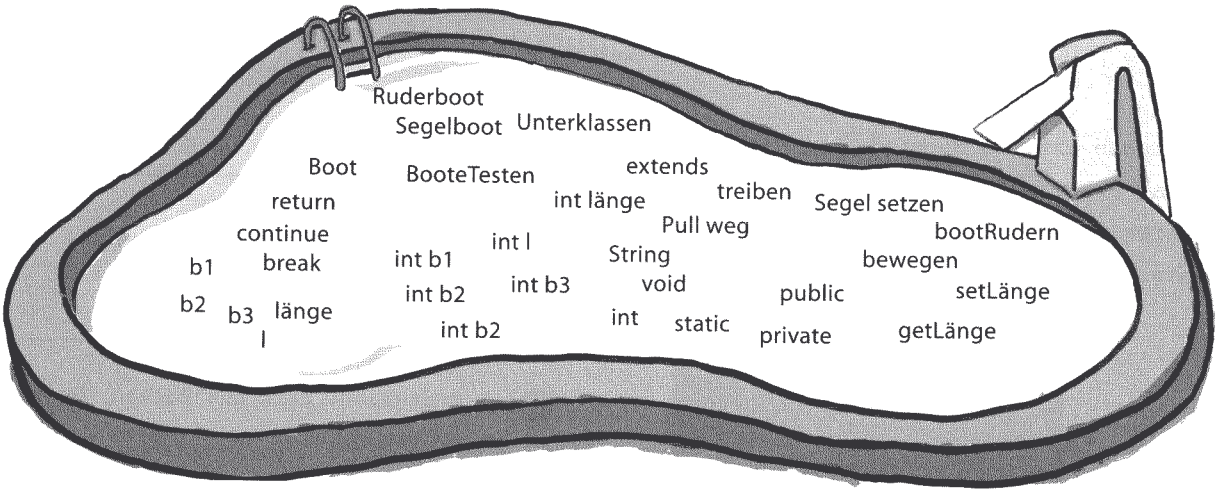
```
public class Ruderboot _________ _________ {
  public _____________ bootRudern() {
    System.out.print("Pull weg");
  }
}
```

```
public class _________ {
  private int _____________ ;
  ________ void _____________ ( _______ ) {
    länge = l;
  }
  public int getLänge() {
    _________ _____________ ;
  }
  public _____________ bewegen() {
    System.out.print("_____________");
  }
}
```

```
public class BooteTesten {
  _______ _______ _________ main(String[] args){
    _________ b1 = new Boot();
    Segelboot b2 = new _____________();
    Ruderboot _________ = new Ruderboot();
    b2.setLänge(32);
    b1._____________();
    b3._____________();
    _________.bewegen();
  }
}
```

```
public class _____________ _________ Boot {
  public ________ _____________() {
    System.out.print("_____________");
  }
}
```

**Ausgabe:** `treiben treiben Segel setzen`



# Spielen Sie Compiler

Satz 1 funktioniert.

## Lösungen zu den Übungen

Satz 2 lässt sich wegen des Rückgabetyps der erschrecken()-Methode von Vampir (int) nicht kompilieren.

Die erschrecken()-Methode von Vampir (B) ist keine gültige Überschreibung ODER Überladung der erschrecken()-Methode von Monster. NUR den Rückgabetyp zu ändern reicht nicht aus, um eine gültige Überladung daraus zu machen. Und da int mit boolean nicht kompatibel ist, ist die Methode keine gültige Überschreibung. (Denken Sie daran, dass Sie einen Rückgabetyp brauchen, der mit dem Rückgabetyp der Superklasse kompatibel ist, wenn Sie NUR den Rückgabetyp ändern, und dass Sie dann eine Über*schreibung* haben.)

Die Sätze 3 und 4 lassen sich kompilieren, erzeugen aber:

```
Huh

Feuerhauch

Huh
```

Denken Sie daran, dass die Klasse Vampir die erschrecken()-Methode von Monster nicht über*schrieben* hat. (Die erschrecken()-Methode für Vampir in Satz 4 nimmt ein byte, kein int.)

---

**Code-Kandidaten:**

**Vermischte Nachrichten**

```
b.m1();
c.m2();
a.m3();

c.m1();
c.m2();
c.m3();

a.m1();
b.m2();
c.m3();

a2.m1();
a2.m2();
a2.m3();
```

**Ausgabe:**

```
As m1, As m2, Cs m3, 6
Bs m1, As m2, As m3,
As m1, Bs m2, As m3,
Bs m1, As m2, Cs m3, 13
Bs m1, Cs m2, As m3,
Bs m1, As m2, Cs m3, 6
As m1, As m2, Cs m3, 13
```

```
public class Ruderboot extends Boot {
  public void bootRudern() {
    System.out.print("Pull weg");
  }
}
public class Boot {
  private int länge ;
  public void setLänge ( int l ) {
    länge = l;
  }
  public int getLänge() {
    return länge ;
  }
  public void bewegen() {
    System.out.print("treiben ");
  }
}
```

```
public class BooteTesten {
  public static void main(String[] args){
    Boot b1 = new Boot();
    Segelboot b2 = new Segelboot();
    Ruderboot b3 = new Ruderboot();
    b2.setLänge(32);
    b1.bewegen();
    b3.bewegen();
    b2.bewegen();
  }
}
public class Segelboot extends Boot {
  public void bewegen() {
    System.out.print("Segel setzen ");
  }
}
```

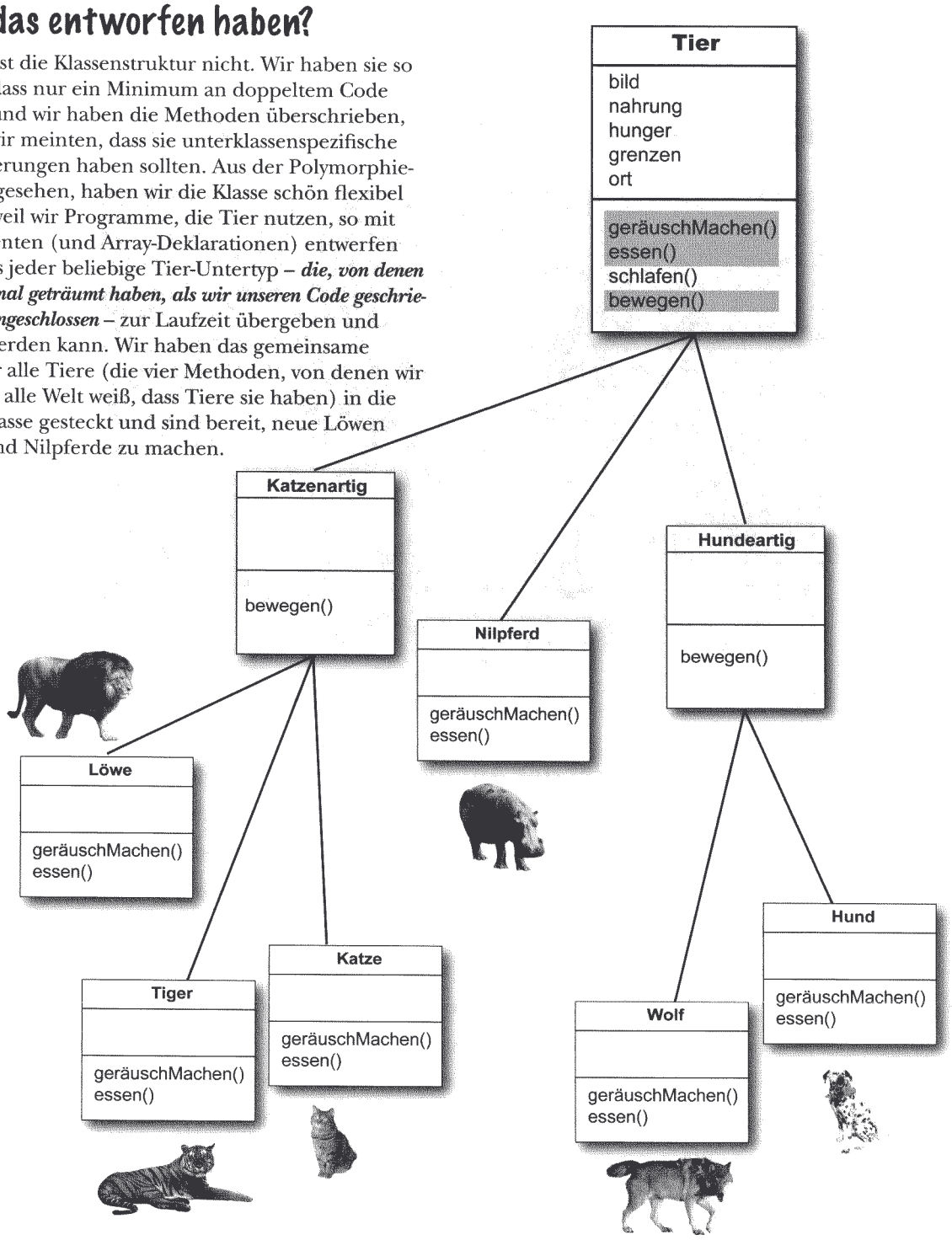
**AUSGABE:** `treiben treiben Segel setzen`

# Ernsthafte Polymorphie

**Vererbung ist erst der Anfang.** Um Polymorphie richtig zu nutzen, brauchen wir Schnittstellen (und zwar nicht die GUI-Typen). 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. Ohne Schnittstellen wären einige der coolsten Teile von Java nicht einmal möglich. Wenn Sie selbst nicht mit Schnittstellen entwerfen, müssen Sie sie also immer noch nutzen. Aber Sie *sollten* mit ihnen entwerfen. Sie werden mit ihnen entwerfen *müssen*. ***Und Sie werden sich wundern, wie Sie je ohne sie leben konnten.*** Was eine Java-Schnittstelle ist? Es ist eine 100% abstrakte Klasse. Was eine abstrakte Klasse ist? Es ist eine Klasse, die nicht instantiiert werden kann. Wozu soll denn das gut sein? Das werden Sie in ein paar Minuten sehen. Denken Sie an das Ende des letzten Kapitels zurück und rufen Sie sich ins Gedächtnis, wie wir polymorphe Argumente verwendet haben, damit eine Tierarzt-Methode Tier-Unterklassen unterschiedlichster Art verwenden konnte: Das hat gerade so die Oberfläche angekratzt. Schnittstellen sind das ***poly*** in Polymorphie. Das ***ab*** in abstrakt. Das ***Koffein*** in Java.

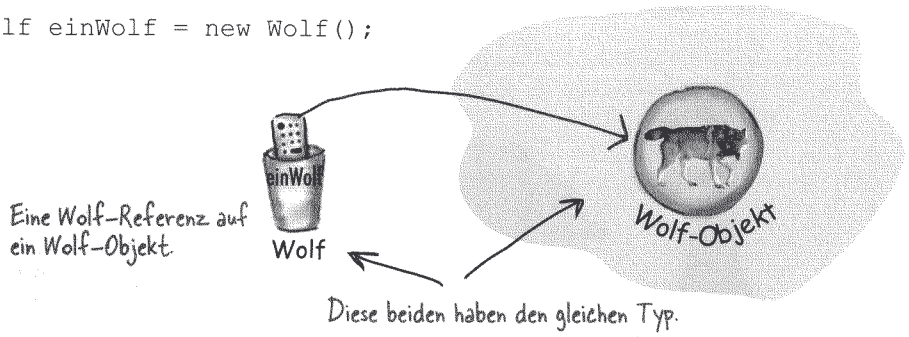
# Haben wir etwas vergessen, als wir das entworfen haben?

So schlecht ist die Klassenstruktur nicht. Wir haben sie so entworfen, dass nur ein Minimum an doppeltem Code vorkommt, und wir haben die Methoden überschrieben, von denen wir meinten, dass sie unterklassenspezifische Implementierungen haben sollten. Aus der Polymorphie-Perspektive gesehen, haben wir die Klasse schön flexibel entworfen, weil wir Programme, die Tier nutzen, so mit Tier-Argumenten (und Array-Deklarationen) entwerfen können, dass jeder beliebige Tier-Untertyp – ***die, von denen wir nicht einmal geträumt haben, als wir unseren Code geschrieben haben, eingeschlossen*** – zur Laufzeit übergeben und verwendet werden kann. Wir haben das gemeinsame Protokoll für alle Tiere (die vier Methoden, von denen wir wollten, dass alle Welt weiß, dass Tiere sie haben) in die Tier-Superklasse gesteckt und sind bereit, neue Löwen und Tiger und Nilpferde zu machen.



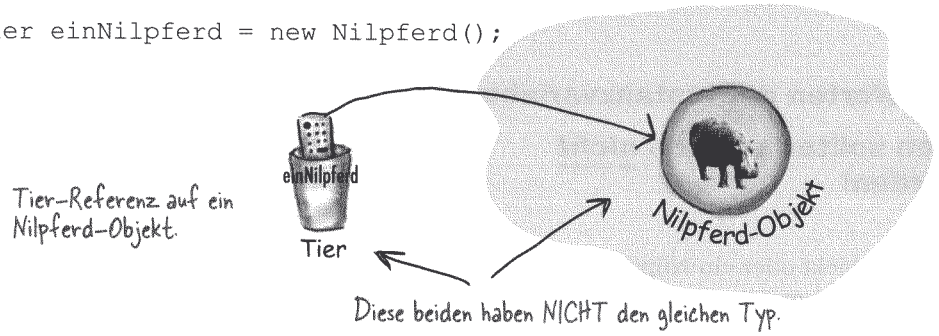
**Wir wissen, dass wir Folgendes sagen können:**

```
Wolf einWolf = new Wolf();
```



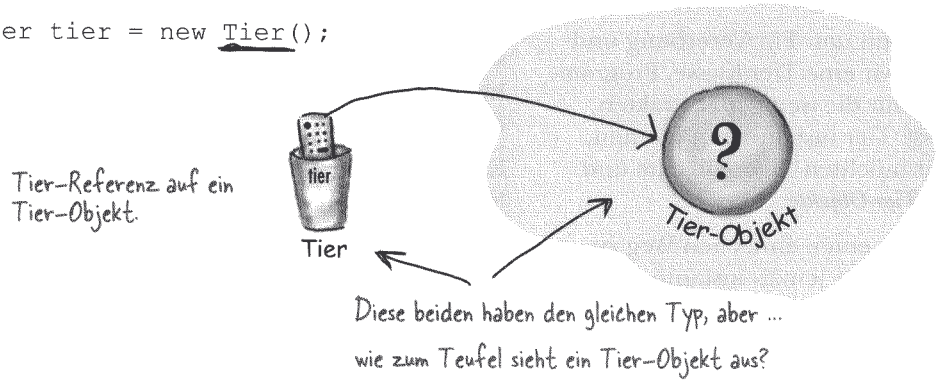

**Und wir wissen, dass wir dieses sagen können:**

```
Tier einNilpferd = new Nilpferd();
```

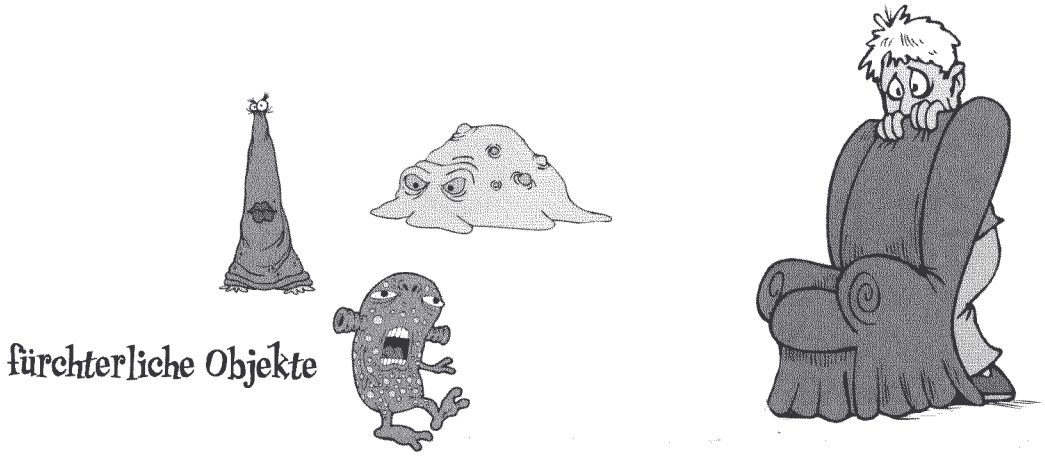



**Aber hier fängt es an, seltsam zu werden:**

```
Tier tier = new Tier();
```



## Wie *sieht* ein neues Tier()-Objekt *aus*?




### Was ist mit den Werten der Instanzvariablen?

### Manche Klassen sollten einfach *nicht* instantiiert werden!

Es ist sinnvoll, ein Wolf-Objekt oder ein Nilpferd-Objekt oder ein Tiger-Objekt zu erzeugen, aber was genau *ist* ein Tier-Objekt? Welche Gestalt hat es? Welche Farbe, Größe, wie viele Beine ...

Der Versuch, ein Objekt des Typs Tier zu erzeugen, ist wie **ein albtraumartiger Star Trek™-Transporter-Unfall.** Der, bei dem irgendwo während des Beamens plötzlich etwas Seltsames mit dem Zwischenspeicher geschieht.

Aber wie gehen wir damit um? Für Vererbung und Polymorphie *brauchen* wir eine Tier-Klasse. Programmierer sollen jedoch nur die weniger abstrakten *Unterklassen* der Klasse Tier instantiieren können, nicht Tier selbst. Wir möchten Tiger-Objekte und Löwe-Objekte, ***keine Tier-Objekte.***

Glücklicherweise gibt es einen einfachen Weg zu verhindern, dass eine Klasse jemals instantiiert wird, oder, anders gesagt, zu verhindern, dass irgendjemand auf diesem Typ »**new**« sagt. Wenn wir die Klasse als **abstract** markieren, hält der Compiler jeden Code davon ab, irgendwo eine Instanz dieses Typs zu erzeugen.

Sie können den abstrakten Typ immer noch als Referenztyp verwenden. Und das ist ein wichtiger Punkt dafür, überhaupt abstrakte Klassen zu machen (um sie als polymorphes Argument oder polymorphen Rückgabetyp zu verwenden oder ein polymorphes Array zu erstellen).

Wenn Sie die Vererbungsstruktur Ihrer Klassen entwerfen, müssen Sie entscheiden, welche Klassen *abstrakt* und welche *konkret* sind. Konkrete Klassen sind die Klassen, die so spezifisch sind, dass sie instantiiert werden können. Dass eine Klasse *konkret* ist, heißt nur, dass es in Ordnung ist, wenn man Objekte diese Typs erstellt.

Es ist einfach, eine abstrakte Klasse zu machen – stellen Sie lediglich das Schlüsselwort **abstract** vor die Klassendefinition:

```
abstract class Hundeartig extends Tier {
   public void bewegen() { }
}
```

# Der Compiler verhindert, dass Sie eine abstrakte Klasse instantiieren

Wenn eine Klasse abstrakt ist, heißt das, dass niemand jemals eine Instanz von dieser Klasse erstellen kann. Für die Zwecke der Polymorphie können Sie die abstrakte Klasse immer noch als einen deklarierten Referenztyp verwenden, aber Sie müssen nicht befürchten, dass jemand Objekte dieses Typs erstellt. Das *garantiert* der Compiler.

```
abstract public class Hundeartig extends Tier
{
   public void bewegen() { }
}
```

---

```
public class MachHundeartig{
   public void los() {
      Hundeartig h;
      h = new Hund();
      h = new Hundeartig();
      h.bewegen();
   }
}
```

Das ist in Ordnung, weil Sie einer Superklassen-Referenz immer ein Unterklassen-Objekt zuweisen können, selbst wenn die Superklasse abstrakt ist.

Die Klasse Hundeartig ist als abstract markiert, der Compiler lässt Sie das also NICHT tun.

```
Datei Bearbeiten Fenster Hilfe BeamMeUp
% javac MachHundeartig.java
MachHundeartig.java:5: Hundeartig is abstract;
cannot be instantiated
      h = new Hundeartig ();
          ^
1 error
```

Eigentlich* hat eine **abstrakte Klasse** keine Verwendung, keinen Wert, keinen Lebenszweck, wenn sie nicht ***erweitert*** wird.

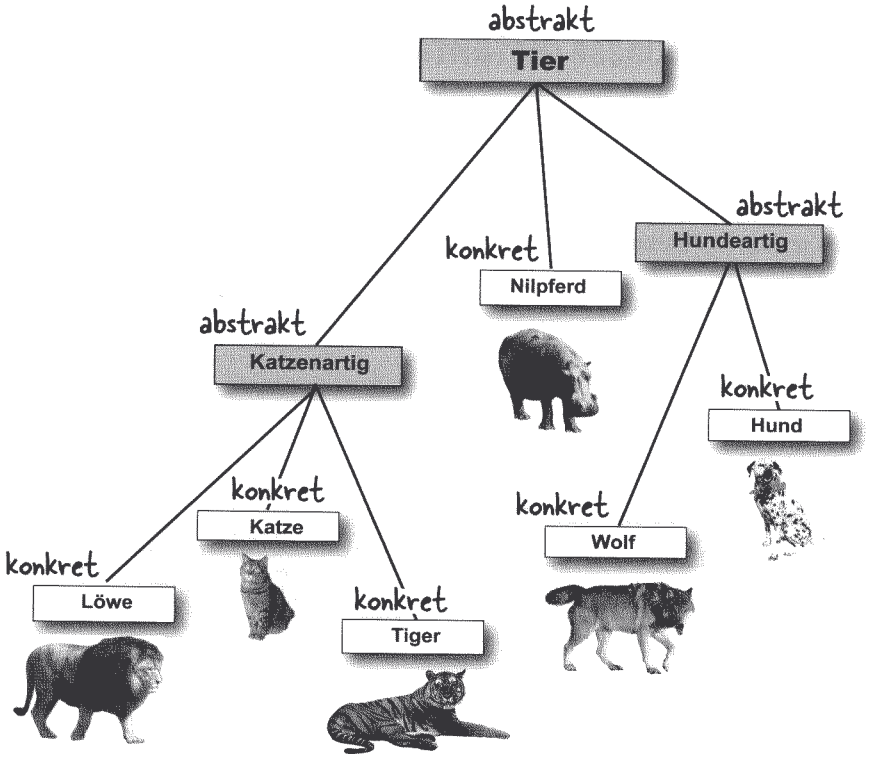
Bei einer abstrakten Klasse sind die Typen, die zur Laufzeit die Arbeit erledigen, die **Instanzen einer *Unterklasse*** Ihrer abstrakten Klasse.

*Es gibt eine Ausnahme – eine abstrakte Klasse kann statische Member haben (siehe Kapitel 10).

# Abstrakt vs. konkret

Eine Klasse, die nicht abstrakt ist, wird als eine *konkrete* Klasse bezeichnet. Wenn wir im Tier-Vererbungsbaum Tier, Hundeartig und Katzenartig abstrakt machen, bleiben Nilpferd, Wolf, Hund, Tiger, Löwe und Katze konkrete Unterklassen.

Blättern Sie die Java-API durch, finden Sie viele abstrakte Klassen, insbesondere in der GUI-Bibliothek. Wie sieht eine GUI-Komponente aus? Die Klasse Component ist die Superklasse der GUI-bezogenen Klassen für Dinge, beispielsweise Buttons, Textfelder, Scrollleisten, Dialogfenster, die Sie verwenden. Sie machen keine Instanz einer generischen *Component* und setzen diese auf den Bildschirm, Sie machen einen JButton. Anders gesagt, Sie instantiieren nur eine *konkrete Unterklasse* von Component, niemals Component selbst.




## KOPF-NUSS

### Abstrakt oder konkret?

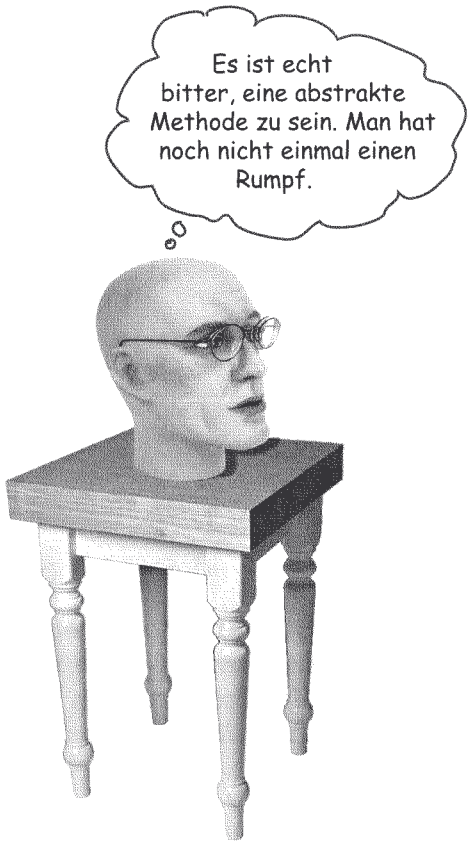
Wie merken Sie, dass eine Klasse abstrakt sein sollte? ***Wein*** ist wahrscheinlich abstrakt. Aber was ist mit ***Rot*** und ***Weiß***? Auch die sind wahrscheinlich abstrakt (zumindest für einige von uns). Aber an welchem Punkt in der Hierarchie werden die Dinge konkret?

Machen Sie ***PinotNoir*** konkret, oder ist auch das abstrakt? »1997 Camelot Vineyards Pinot Noir« ist wahrscheinlich konkret, egal was es ist. Aber wie können Sie da sicher sein?

Sehen Sie sich den Tier-Vererbungsbaum oben an. Scheint Ihnen unsere Entscheidung darüber vernünftig, welche Klassen abstrakt und welche konkret sein sollen? Würden Sie am Tier-Vererbungsbaum etwas ändern (außer mehr Tiere hinzuzufügen natürlich)?

# Abstrakte Methoden

Außer Klassen kann man auch *Methoden* als abstract markieren. Wenn eine Klasse abstrakt ist, heißt das, dass die Klasse *erweitert* werden muss. Ist eine Methode abstrakt, bedeutet das, dass die Methode *überschrieben* werden muss. Vielleicht entscheiden Sie, dass einige (oder alle) Verhalten in einer abstrakten Klasse sinnlos sind, wenn sie nicht von einer konkreteren Unterklasse implementiert werden. Anders gesagt, Sie können sich keine generische Methodenimplementierung vorstellen, die für die Unterklassen nützlich sein könnte. Wie sähe eine generische *essen()*-Methode aus?




## Eine abstrakte Methode hat keinen Body!

Weil Sie bereits entschieden haben, dass es keinen Code gibt, der in der abstrakten Methode sinnvoll wäre, geben Sie keinen Methoden-Body an. Es gibt also keine geschweiften Klammern – Sie beenden die Deklaration einfach mit einem Semikolon.

```
public abstract void essen();
```

Kein Methoden-Body! Endet mit einem Semikolon.

**Wenn Sie eine abstrakte *Methode* deklarieren, MÜSSEN Sie die *Klasse* ebenfalls als abstract markieren. In einer nicht abstrakten Klasse dürfen keine abstrakten Methoden auftauchen.**

Stecken Sie in eine Klasse eine einzige abstrakte Methode, müssen Sie die Klasse abstrakt machen. Aber Sie *können* in einer Klasse abstrakte und nicht abstrakte Methoden mischen.

## Es gibt keine Dummen Fragen

**F: Was ist der *Sinn* einer abstrakten Methode? Ich dachte, dass es bei abstrakten Klassen gerade darum geht, gemeinsamen Code zu haben, der von den Unterklassen geerbt werden kann.**

A: Es ist eine gute Sache, wenn man vererbbare Methodenimplementierungen (d.h. Methoden mit einem richtigen *Body*) in eine Superklasse steckt. *Wenn das sinnvoll ist.* Und in einer abstrakten Klasse ist das oft *nicht sinnvoll*, weil Sie keinen generischen Code entwerfen können, der für die Unterklassen nützlich wäre. Der Sinn einer abstrakten Methode ist, dass Sie, auch wenn Sie keinen Methodencode geschrieben haben, trotzdem noch einen Teil des *Protokolls* für eine Gruppe von Untertypen (Unterklassen) definiert haben.

**F: Und das ist gut, weil ...**

A: Polymorphie! Denken Sie daran, dass Sie nach der Möglichkeit suchen, einen (oft abstrakten) Supertyp als Methodenargument, Rückgabetyp oder Array-Typ zu verwenden. Auf diese Weise können Sie Ihrem Programm neue Untertypen (wie eine neue Tier-Unterklasse) hinzufügen, ohne Methoden umschreiben (oder neue hinzufügen) zu müssen, die mit diesen neuen Typen umgehen können. Stellen Sie sich vor, Sie müssten die Klasse Tierarzt ändern, wenn sie nicht Tier als Argumenttyp für ihre Methoden verwenden würde. Sie bräuchten eine separate Methode für jede einzelne Tier-Unterklasse! Eine, die einen Löwen nimmt, eine, die einen Wolf nimmt, eine, die ... Sie verstehen schon. Mit einer abstrakten Methode sagen Sie also im Sinne der Polymorphie: »Alle Untertypen dieses Typs haben DIESE Methode.«

# Sie MÜSSEN alle abstrakten Methoden implementieren

**Eine abstrakte Methode *implementieren* ist das Gleiche wie das *Überschreiben* einer Methode.**

Abstrakte Methoden haben keinen Body. Es gibt sie nur für die Polymorphie. Das bedeutet, dass die erste konkrete Klasse im Vererbungsbaum *alle* abstrakten Methoden implementieren muss.

Sie können den Kelch weiterreichen, indem Sie selbst abstrakt bleiben. Wenn sowohl Tier als auch Hundeartig abstrakt sind und beide abstrakte Methoden haben, muss die Klasse Hundeartig die abstrakten Methoden von Tier beispielsweise nicht implementieren. Aber sobald wir zur ersten konkreten Unterklasse wie Hund kommen, muss diese *alle* abstrakten Methoden von Tier und Hundeartig implementieren.

Aber denken Sie daran, dass eine abstrakte Klasse gleichzeitig abstrakte und *nicht* abstrakte Methoden haben kann. Hundeartig könnte also beispielsweise eine abstrakte Methode von Tier implementieren, damit Hund das nicht mehr tun muss. Aber wenn Hundeartig nichts zu den abstrakten Methoden von Tier sagt, muss Hund alle abstrakten Methoden von Tier implementieren.

Wenn wir sagen: »Sie müssen die abstrakte Methode implementieren«, heißt das, dass Sie einen *Methoden-Body liefern müssen*. Sie müssen also in Ihrer Klasse eine nicht abstrakte Methode mit der gleichen Methodensignatur (Name und Argumente) und einem Rückgabetyp angeben, der mit dem deklarierten Typ der abstrakten Methode kompatibel ist. Was Sie *in* diese Methode packen, bleibt Ihnen überlassen. Java interessiert nur eins: dass die Methode in Ihrer konkreten Unterklasse *vorhanden* ist.

## Abstrakte gegen konkrete Klassen

Bringen wir diese ganze abstrakte Rhetorik mal zur Anwendung. In der mittleren Spalte haben wir einige Klassen aufgeführt. Ihre Aufgabe ist es, sich auf der einen Seite Anwendungen vorzustellen, in denen die aufgeführte Klasse eine konkrete Klasse sein könnte, und auf der anderen Seite Anwendungen, in denen die aufgeführte Klasse eine abstrakte Klasse sein könnte. Wir haben uns die ersten Anwendungen schon mal vorgenommen, um Ihnen auf die Sprünge zu helfen. In einem Baumschulen-Programm könnte die Klasse Baum beispielsweise abstrakt sein, weil der Unterschied zwischen einer Eiche und einer Esche eine Rolle spielt. Aber in einer Golf-Simulation könnte Baum eine konkrete Klasse sein (beispielsweise eine Unterklasse von Hindernis), weil das Programm nicht zwischen verschiedenen Typen von Bäumen unterscheidet. (Es gibt keine richtige Antwort, denn sie ist von Ihrem Entwurf abhängig.)

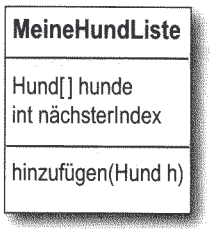
| Konkret | Beispielklasse | Abstrakt |
|---|---|---|
| Golfsimulation | Baum | Baumschulen-Programm |
| _____ | Haus | Architektur-Anwendung |
| Satellitenfoto-Anwendung | Stadt | _____ |
| _____ | Fußballspieler | Trainingsprogramm |
| _____ | Stuhl | _____ |
| _____ | Kunde | _____ |
| _____ | Bestellung | _____ |
| _____ | Buch | _____ |
| _____ | Geschäft | _____ |
| _____ | Lieferant | _____ |
| _____ | Golfclub | _____ |
| _____ | Motor | _____ |
| _____ | Herd | _____ |

# Polymorphie in Aktion

Nehmen wir an, wir möchten unsere *eigene* Listenklasse schreiben, eine, die Hund-Objekte festhält, und geben für den Augenblick vor, dass wir die Klasse ArrayList nicht kennen. Beim ersten Versuch geben wir ihr nur eine *hinzufügen()*-Methode. Wir verwenden ein einfaches Hund-Array (Hund []), um die hinzugefügten Hund-Objekte festzuhalten, und geben ihm die Länge 5. Wenn wir die Grenze von 5 Hund-Objekten erreichen, können Sie die Methode *hinzufügen()* immer noch aufrufen. Aber dann macht sie nichts. Haben wir die Grenze *noch* nicht erreicht, steckt die Methode *hinzufügen()* den Hund an der nächsten verfügbaren Indexposition (nächsterIndex) in das Array und inkrementiert dann die Variable für den nächsten verfügbaren Index.

## Eine eigene Hund-spezifische Liste erstellen

**(Vielleicht der schlechteste Versuch aller Zeiten, eine eigene ArrayList-artige Klasse aus dem Boden zu stampfen.)**




```
public class MeineHundListe {

    private Hund [] hunde = new Hund[5];

    private int nächsterIndex = 0;


    public void hinzufügen(Hund h) {

        if (nächsterIndex < hunde.length) {

            hunde[nächsterIndex] = h;

            System.out.println("Hund eingefügt bei " + nächsterIndex);

            nächsterIndex++;

        }

    }

}
```

Im Hintergrund ein gewöhnliches Hund-Array verwenden.

Wird jedes Mal inkrementiert, wenn ein neuer Hund hinzugefügt wird.

Wenn wir die Grenze des Hund-Arrays noch nicht erreicht haben, den Hund hinzufügen und eine Nachricht ausgeben.

Inkrementieren, um dem Benutzer den nächsten Index zur Verfügung zu stellen.

# Oh nein, jetzt müssen wir auch noch Katzen halten.

Uns bieten sich jetzt verschiedene Möglichkeiten:

1) Eine separate Klasse, MeineKatzeListe, erstellen, die Katze-Objekte speichert. Ziemlich klobig.

2) Eine einzelne Klasse, HundUndKatzeListe, machen, die zwei verschiedene Array-Instanzvariablen und zwei verschiedene hinzufügen()-Methoden, katzeHinzufügen(Katze k) und hundHinzufügen(Hund h), hat. Noch eine klobige Lösung.

3) Eine heterogene Klasse TierListe erzeugen, die *jede* Art Tier-Unterklasse nimmt (schließlich wissen wir, dass sich die Spezifikation schon einmal geändert hat, um Platz für Katzen zu machen, und können uns denken, dass wir früher oder später auch noch irgendeine *andere* Art Tier hinzufügen müssen). Uns gefällt diese Option am besten. Machen wir also unsere Klasse allgemeiner, damit sie Tiere statt nur Hunde annimmt. Wir haben die wichtigsten Änderungen hervorgehoben (die Logik bleibt natürlich die gleiche, aber im ganzen Code hat sich der Typ von Hund in Tier geändert).

## Unsere eigene Tier-spezifische Liste implementieren

Version 2

```
public class MeineTierListe {

    private Tier[] tiere = new Tier[5];
    private int nächsterIndex = 0;

    public void hinzufügen(Tier t) {
       if (nächster Index < tiere.length) {
          tiere[nächsterIndex] = t;
          System.out.println("Tier hinzugefügt bei " + nächsterIndex);
          nächsterIndex++;
       }
    }
}
```

Keine Panik. Wir machen kein neues Tier-Objekt. Wir machen ein neues Array-Objekt vom Typ Tier. (Denken Sie daran, Sie können keine Instanz eines abstrakten Typs erstellen, aber Sie KÖNNEN ein Array-Objekt machen, das deklarationsgemäß diesen Typ AUFNIMMT.)

---

```
public class TierTestlauf{
   public static void main (String[] args) {
      MeineTierListe liste = new MeineTierListe();
      Hund a = new Hund();
      Katze b = new Katze();
      liste.hinzufügen(a);
      liste.hinzufügen(b);
   }
}
```

```
Datei Bearbeiten Fenster Hilfe Verletzen
% java TierTestlauf
Tier hinzugefügt bei 0
Tier hinzugefügt bei 1
```

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

Sie wissen, wohin das führt. Wir möchten den Typ des Arrays und der *hinzufügen()*-Methode so ändern, dass sie mit etwas arbeitet, das *über* Tier steht. Etwas, das sogar *noch* allgemeiner, *noch* abstrakter ist als Tier. Aber wie können wir das tun? Wir *haben* keine Superklasse für Tier.

Version 3

(Das sind nur ein paar der Methoden in ArrayList ... es gibt noch viel mehr.)

Moment mal, vielleicht haben wir das doch ...

Erinnern Sie sich an all diese Methoden von ArrayList? Sehen Sie, die Methoden remove(), contains() und indexOf() verwenden alle Objekte eines einzigen Typs ... ***Object!***

**ArrayList**

**boolean remove(Object elem)**
Entfernt das durch den Parameter angegebene Objekt. Liefert true zurück, wenn das Element in der Liste enthalten war.

**boolean contains(Object elem)**
Liefert true zurück, wenn die Liste ein Objekt enthält, das dem durch den Parameter angegebenen Objekt entspricht.

**boolean isEmpty()**
Liefert true zurück, wenn die Liste leer ist.

**int indexOf(Object elem)**
Liefert entweder die Indexposition des durch den Parameter angegebenen Objekts oder –1 zurück.

**Object get(int index)**
Liefert das Element zurück, das sich an der durch den Parameter angegebenen Indexposition befindet.

**boolean add(Object elem)**
Fügt der Liste das Element hinzu (liefert true zurück).

**// mehr**

Viele der Methoden von ArrayList nutzen den ultimativen polymorphen Typ Object. Da in Java jede Klasse eine Unterklasse von Object ist, können diese Methoden von ArrayList alles entgegennehmen!

(Hinweis: Seit Java 5.0 sehen die Methoden get() und add() eigentlich etwas anders aus als die, die wir hier zeigen. Aber für den Augenblick stellen wir sie uns so vor. Die ganze Geschichte gibt es etwas später.)

## Jede Java-Klasse erweitert die Klasse Object.

Die Klasse Object ist die Mutter aller Klassen. Sie ist die Superklasse von *allem*. Selbst wenn Sie die Vorteile der Polymorphie ausnutzen, müssen Sie immer noch eine Klasse mit Methoden erstellen, die *Ihren* polymorphen Typ entgegennehmen und zurückliefern. Gäbe es in Java keine gemeinsame Superklasse für alles, gäbe es für Entwickler auch keine Möglichkeit, Klassen mit Methoden zu erstellen, die *Ihre* Typen entgegennehmen ... *Typen, von denen die Entwickler nicht einmal träumten, als sie die Klasse ArrayList schrieben.*

Also machen wir schon von Anfang an Unterklassen der Klasse Object, ohne dass wir das wussten. ***Jede Klasse, die Sie schreiben, erweitert Object,*** ohne dass Sie das irgendwo sagen müssten. Aber Sie können sich das so vorstellen, als sähe eine von Ihnen geschriebene Klasse folgendermaßen aus:

```
public class Hund extends Object { }
```

Einen Augenblick bitte, Hund erweitert doch *schon* etwas, *Hundeartig*. Das ist in Ordnung. Der Compiler lässt stattdessen *Hundeartig* Object erweitern. Aber *Hundeartig* erweitert doch Tier. Kein Problem, dann lässt der Compiler eben einfach *Tier* Object erweitern.

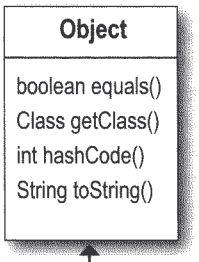
## Jede Klasse, die nicht *explizit* eine andere Klasse erweitert, erweitert *implizit* Object.

Da Hund Hundeartig erweitert, erweitert es Object nicht *direkt* (auch wenn es Object indirekt erweitert). Und das Gleiche gilt ebenfalls für Hundeartig. Aber Tier *erweitert* Object direkt.

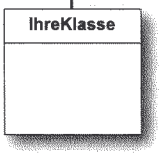
# Was also steckt in dieser Ultra-Super-Mega-Klasse Object?

Wenn Sie Java wären, welches Verhalten sollte dann *jedes* Objekt haben? Hmmmm ... schaun wir mal ... wie wäre es mit einer Methode, mit der man herausfinden kann, ob ein Objekt gleich einem anderen Objekt ist? Und was wäre mit einer Methode, die einem den tatsächlichen Typ des Objekts sagen kann? Vielleicht eine Methode, die einem den Hashcode für das Objekt gibt, damit man es in einer Hashtable verwenden kann (über Javas Hashtables werden wir in Kapitel 16 sprechen)? Ach ja, und das wäre gut – eine Methode, die eine String-Nachricht für das Objekt ausgibt.

Und wissen Sie was? Wie durch Zauberei hat die Klasse Object tatsächlich Methoden für diese vier Dinge. Das ist zwar nicht alles, aber das sind die Methoden, für die wir uns wirklich interessieren.




Nur EINIGE der Methoden der Klasse Object.




Jede Klasse, die Sie schreiben, erbt diese Methoden von Object. Die Klassen, die Sie bisher geschrieben haben, haben Methoden geerbt, von denen Sie nicht einmal wussten, dass es sie gibt.

## ① equals(Object o)

```
Hund h = new Hund();
Katze k = new Katze();

if (h.equals(k)) {
    System.out.println("wahr");
} else {
    System.out.println("falsch");
}
```

```
Datei Bearbeiten Fenster Hilfe Stopp
% java TestObject
falsch
```

Sagt Ihnen, ob zwei Objekte als gleich (equal) betrachtet werden. (Was »gleich« wirklich heißt, werden wir uns in Anhang B ansehen.)

## ② getClass()

```
Katze k = new Katze();
System.out.println(k.getClass());
```

```
Datei Bearbeiten Fenster Hilfe Ohnmacht
% java TestObject
class Katze
```

Liefert Ihnen die Klasse, von der dieses Objekt instantiiert wurde.

## ③ hashCode()

```
Katze k = new Katze();
System.out.println(k.hashCode());
```

```
Datei Bearbeiten Fenster Hilfe Hinfallen
% java TestObject
8202111
```

Gibt den Hashcode für das Objekt aus. (Im Augenblick können Sie das als eine Art eindeutige ID betrachten.)

## ④ toString()

```
Katze k = new Katze();
System.out.println(k.toString());
```

```
Datei Bearbeiten Fenster Hilfe InsKomaFallen
% java TestObject
Katze@7d277f
```

Gibt eine String-Nachricht mit dem Namen der Klasse und einer anderen Zahl aus, die uns echt interessiert.

## Es gibt keine Dummen Fragen

**F: Ist die Klasse Object abstrakt?**

A: Nein. Zumindest nicht im formalen Java-Sinn. Object ist eine nicht abstrakte Klasse, weil sie Methodenimplementierungscode besitzt, den alle Klassen erben und den sie verwenden können, ohne die Methoden überschreiben zu müssen.

**F: Es ist also *möglich*, die Methoden in Object zu überschreiben?**

A: Einige von ihnen. Aber ein paar sind als `final` markiert, und das bedeutet, dass Sie sie nicht überschreiben können. Es wird Ihnen (dringlichst) ans Herz gelegt, hashCode(), equals() und toString() in Ihren eigenen Klassen zu überschreiben, und etwas später in diesem Buch werden Sie auch lernen, wie Sie das tun. Aber einige Methoden wie getClass() machen Dinge, die auf eine bestimmte, gesicherte Weise arbeiten müssen.

**F: Wenn die Methoden von ArrayList allgemein genug sind, um Object zu verwenden, was bedeutet es dann, wenn man ArrayList<DotCom> sagt? Ich dachte, damit würde ich die ArrayList so einschränken, dass sie nur noch DotCom aufnehmen kann.**

A: Sie *haben* sie eingeschränkt. Vor Java 5.0 konnten ArrayLists nicht eingeschränkt werden. Im Grunde waren sie vollkommen mit dem identisch, was Sie heute in Java 5.0 erhalten, wenn Sie ArrayList<Object> schreiben. Anders gesagt, ***eine ArrayList, die auf alles eingeschränkt ist, was ein Object ist,*** und das bedeutet in Java *jedes* Objekt, das von *einer beliebigen* Klasse instantiiert wurde! Die Details zu dieser neuen <Typ>-Syntax werden wir später im Buch behandeln.

**F: Gut, kommen wir noch einmal darauf zurück, dass die Klasse Object nicht abstrakt ist (was, wie ich annehme, heißt, dass sie konkret ist). WIE kann man jemanden ein Objekt vom Typ Object machen lassen? Ist das nicht genauso seltsam, wie ein Objekt vom Typ Tier zu machen?**

A: Gute Frage! Warum es akzeptabel ist, eine neue Object-Instanz zu machen? Weil man manchmal ein generisches Object als ein ... ein Objekt eben verwenden möchte. Ein *leichtgewichtiges* Objekt. Bei Weitem am häufigsten werden Instanzen vom Typ Object zur Thread-Synchronisierung verwendet (über die Sie in Kapitel 15 etwas erfahren werden). Schieben Sie das für den Moment auf die Warmhalteplatte und gehen Sie davon aus, dass man selten Objekte vom Typ Object macht, obwohl man es machen *könnte*.

**F: Kann man also sagen, dass der Hauptzweck für den Typ Object ist, dass man ihn als polymorphes Argument und als polymorphen Rückgabetyp verwenden kann? Wie in ArrayList?**

A: Die Klasse Object hat *zwei* wichtige Existenzgründe: Sie dient als polymorpher Typ für Methoden, die mit allen Klassen arbeiten können müssen, die Sie oder irgendein anderer erstellen, und sie bietet *echten* Methodencode, den alle Objekte in Java zur Laufzeit benötigen (und der automatisch von allen Klassen geerbt wird, wenn er in die Klasse Object gesteckt wird). Einige der wichtigsten Methoden in Object beziehen sich auf Threads. Diese werden wir später im Buch kennen lernen.

**F: Wenn es so gut ist, polymorphe Typen zu verwenden, warum lässt man dann nicht einfache ALLE Methoden Object als Argument- und Rückgabetyp verwenden?**

A: Ähhhh ... überlegen Sie mal, was dann passieren würde. Erstens würde man damit den Sinn der »Typsicherheit« vernichten, und die ist einer von Javas wichtigsten Schutzmechanismen für Ihren Code. Mit Typsicherheit garantiert Java, dass Sie nicht das falsche Objekt nach etwas fragen, das Sie *eigentlich* von einem anderen Objekt haben wollten. Beispielsweise einen *Ferrari* (den Sie für einen *Toaster* halten) bitten, sich *selbst* zu kochen. Aber die Wahrheit ist, dass Sie sich um dieses feurige Ferrari-Szenario *keine* Gedanken machen müssen, selbst *wenn* Sie Object-Referenzen für alles verwenden. Denn wenn ein Objekt durch eine Referenz vom Typ Object referenziert wird, *denkt* Java, dass diese Referenz eine Instanz des Typs Object referenziert. Und das bedeutet, dass Sie auf diesem Objekt nur die Methoden aufrufen dürfen, die in der Klasse Object deklariert sind! Nehmen wir an, Sie würden sagen:

```
Object o = new Ferrari();
o.fahrSchnell();
//Nicht erlaubt!
```

Sie würden nicht einmal am Compiler vorbeikommen. Weil Java eine streng typisierte Sprache ist, prüft der Compiler den Typ, um sicherzustellen, dass Sie eine Methode auf einem Objekt aufrufen, das tatsächlich dazu in der Lage ist *zu antworten*. Das bedeutet, Sie können auf einer Objektreferenz nur Methoden aufrufen, *wenn* die Klasse des Referenztyps diese Methode *besitzt*. Das werden wir uns bald noch viel ausführlicher ansehen. Machen Sie sich also keine Gedanken, wenn das Bild jetzt noch nicht kristallklar ist.

# Die Verwendung polymorpher Referenzen vom Typ Object hat ihren Preis ...

Bevor Sie losrennen und anfangen, den Typ Object für Ihre ultraflexiblen Argument- und Rückgabetypen zu verwenden, müssen Sie sich über einen kleinen Haken bei der Verwendung des Typs Object als Referenz Gedanken machen. Und denken Sie auch daran, dass wir nicht davon sprechen, Instanzen vom Typ Object zu machen. Wir sprechen davon, Instanzen von einem anderen Typ zu machen, aber Referenzen vom Typ Object zu verwenden.

Wenn Sie ein Objekt in eine ArrayList<**Hund**> stecken, geht es als Hund rein und kommt als Hund raus:

```
ArrayList<Hund> meineHundArrayList = new ArrayList<Hund>();
Hund einHund = new Hund();
meineHundArrayList.add(einHund);
Hund h = meineHundArrayList.get(0);
```

← Eine ArrayList machen, die Hund-Objekte aufnehmen soll.
← Einen Hund machen.
← Den Hund der Liste hinzufügen.
← Den Hund aus der Liste einer neuen Hund-Referenzvariablen zuweisen. (Denken Sie sich das, als würde die Methode get() den Rückgabetyp Hund deklarieren, weil Sie ArrayList<Hund> verwendet haben.)

Aber was passiert, wenn Sie die Liste als ArrayList<**Object**> deklarierten? Wenn Sie eine ArrayList erstellen wollen, die wirklich *jede* Art von Objekt aufnehmen kann, deklarieren Sie sie folgendermaßen:

```
ArrayList<Object> meineHundArrayList = new ArrayList<Object>();
Hund einHund = new Hund();
meineHundArrayList.add(einHund);
```

← Eine ArrayList machen, die jeden Typ Objekt aufnehmen kann.
← Einen Hund machen.
← Den Hund der Liste hinzufügen.
(Diese zwei Schritte bleiben gleich.)

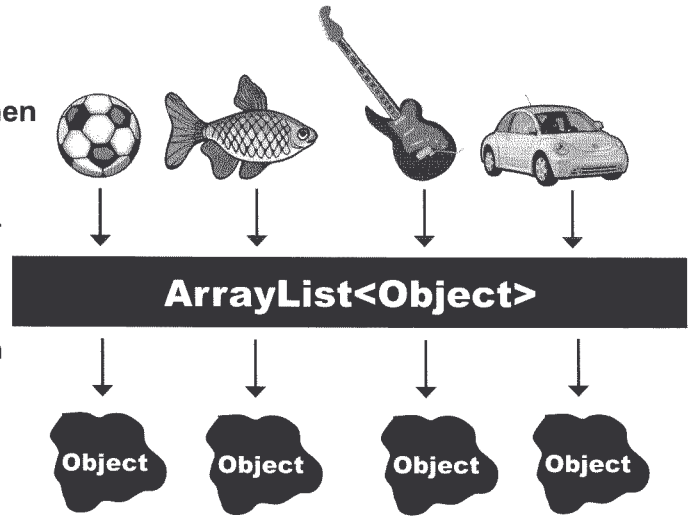
Aber was passiert, wenn Sie versuchen, das Hund-Objekt herauszuholen und einer Hund-Referenz zuzuweisen?

```
Hund h = meineHundArrayList.get(0);
```

NEIN!! Lässt sich nicht kompilieren!! Wenn Sie ArrayList<Object> verwenden, liefert die Methode get() den Typ Object zurück. Der Compiler weiß nur, dass das Objekt (irgendwo in seinem Vererbungsbaum) von Object erbt, aber er weiß nicht, dass es ein Hund ist!!

*Aus einer ArrayList<Object> kommt alles als Referenz vom Typ* ***Object*** *heraus, ganz gleich, was das Objekt eigentlich ist oder welchen Typ die Referenz hatte, als Sie das Objekt der Liste hinzufügten.*

**Die Objekte gehen als Fußball, Fisch, Gitarre und Auto REIN.**




**Aber sie kommen RAUS, als wären sie alle vom Typ Object.**

**Die Objekte, die aus einer ArrayList<Object> herauskommen, verhalten sich alle, als wären sie generische Instanzen der Klasse Object. Der Compiler kann nicht davon ausgehen, dass das Objekt, das herauskommt, einen anderen Typ als Object hat.**

# Wenn ein Hund sich nicht wie ein Hund verhalten will

Es gibt folgendes Problem, wenn man alles polymorph als Object behandelt: Objekte *scheinen* (wenn auch nicht für immer) ihr eigentliches Wesen zu verlieren. *Der Hund scheint sein Hundsein zu verlieren.* Sehen wir uns an, was passiert, wenn wir einer Methode einen Hund übergeben, die das gleiche Hund-Objekt zurückliefert, aber den Rückgabetyp Object an Stelle von Hund deklariert.

SCHLECHT ☹

```
public void los() {
    Hund einHund = new Hund();
    Hund gleicherHund = getObject(einHund);
}

public Object getObject(Object o) {
    return o;
}
```

Diese Zeile funktioniert nicht! Auch wenn die Methode eine Referenz auf denselben Hund liefert, den das Argument referenzierte, bedeutet der Rückgabetyp Object, dass der Compiler Ihnen nicht erlaubt, die zurückgelieferte Referenz irgendetwas anderem zuzuweisen als einer Referenz vom Typ Object.

Wir liefern eine Referenz auf das gleiche Hund-Objekt zurück, aber mit dem Rückgabetyp Object. Dieser Teil ist vollkommen in Ordnung. Hinweis: Auf ähnliche Weise funktioniert die Methode get(), wenn Sie eine ArrayList<Object> statt einer ArrayList<Hund> haben.

```
Datei Bearbeiten Fenster Hilfe DenkenSieDaran
HundPolyTest.java:10: incompatible types
found : java.lang.Object
required: Hund
  Hund gleicherHund = getObject(einHund);
1 error                          ^
```

Der Compiler weiß nicht, dass das Ding, das von der Methode zurückgeliefert wird, eigentlich ein Hund ist, und erlaubt Ihnen deswegen nicht, es einer Hund-Referenz zuzuweisen. (Warum, werden Sie auf der nächsten Seite sehen.)

---

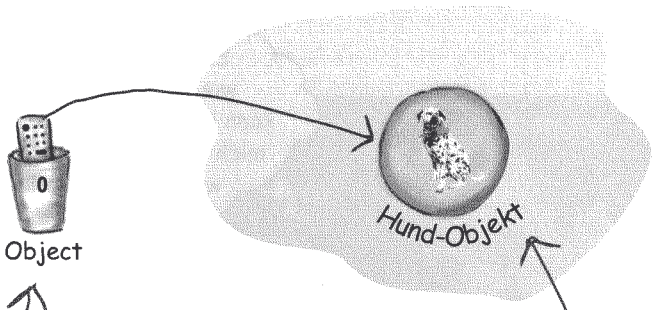
GUT ☺

```
public void los() {
    Hund einHund = new Hund();
    Object gleicherHund = getObject(einHund);
}

public Object getObject(Object o) {
    return o;
}
```

Das funktioniert (auch wenn es, wie Sie gleich sehen werden, nicht sonderlich nützlich ist), weil Sie einer Referenz vom Typ Object ALLES zuweisen können, da jede Klasse einen IS-A-Test für Object besteht. In Java ist jedes Objekt eine Instanz des Typs Object, weil in Java jede Klasse Object an der Spitze ihres Vererbungsbaums hat.

# Object bellt nicht

Jetzt wissen wir also, dass ein Objekt, das von einer Variablen referenziert wird, deren deklarierter Typ Object ist, keiner Variablen zugewiesen werden kann, deren deklarierter Typ der eigentliche Typ des Objekts ist. Und wir wissen, dass das passieren kann, wenn ein deklarierter Rückgabe- oder Argumenttyp Object ist, wie es beispielsweise der Fall ist, wenn das Objekt in eine ArrayList des Typs Object (ArrayList<Object>) gesteckt wird. Aber welche Folgen hat das? Ist es ein Problem, wenn eine Object-Referenzvariable verwendet wird, um ein Hund-Objekt zu referenzieren? Versuchen wir, Hund-Methoden auf unserem Hund-den-Compiler-für-Object-hält aufzurufen:




Wenn Sie eine Objektreferenz von einer ArrayList<Object> (oder einer beliebigen anderen Methode, die als Rückgabetyp Object deklariert) erhalten, kommt diese als polymorphe Referenz des Typs Object an. Sie haben also eine Object-Referenz auf (in diesem Fall) eine Hund-Instanz.

```
Object o = al.get(index);
int i = o.hashCode();
o.bellen();
```

Das ist in Ordnung. Die Klasse Object hat eine hashCode()-Methode. Sie können diese Methode also auf JEDEM Java-Objekt aufrufen.

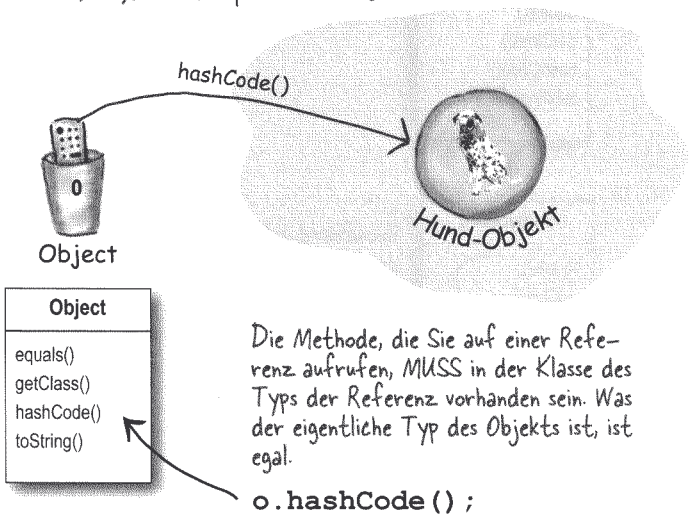
Lässt sich nicht kompilieren!

Das können Sie nicht tun!! Die Klasse Object hat keine Ahnung, was bellen() heißen soll. Selbst wenn SIE wissen, dass sich an dieser Indexposition eigentlich ein Hund befindet, weiß der Compiler das nicht.

**Ob Sie eine Methode aufrufen können, entscheidet der Compiler auf Basis des *Referenz*typs, nicht auf Basis des eigentlichen Typs des *Objekts*.**

Selbst wenn Sie *wissen*, dass das Objekt dazu in der Lage ist (»... es ***ist*** wirklich ein Hund, ehrlich!«), sieht der Compiler es nur als generisches Object. Der Compiler weiß, dass Sie dort auch ein Button-Objekt verstaut haben könnten. Oder ein Mikrowelle-Objekt. Oder irgendetwas anderes, das absolut keine Ahnung davon hat, wie man bellt.

Der Compiler prüft die Klasse des Typs der *Referenz* – nicht den Typ des *Objekts* –, um zu prüfen, ob Sie auf dieser Referenz eine Methode aufrufen können.



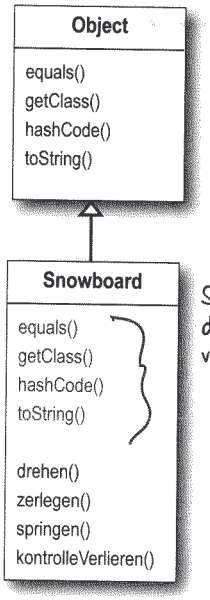

Die Methode, die Sie auf einer Referenz aufrufen, MUSS in der Klasse des Typs der Referenz vorhanden sein. Was der eigentliche Typ des Objekts ist, ist egal.

```
o.hashCode();
```

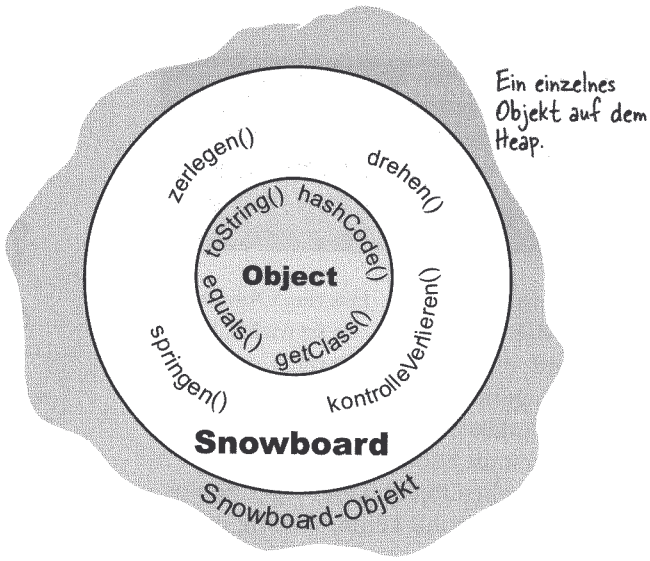
Die Referenz »o« wurde mit dem Typ Object deklariert, Sie können also nur die Methoden aufrufen, die in der Klasse Object sind.

## Finden Sie Ihr inneres Object.

Ein Objekt enthält *alles*, was es von allen seinen Superklassen erbt. Da bedeutet, dass *jedes* Objekt – unabhängig von seinem eigentlichen Typ – *auch* eine Instanz der Klasse Object ist. Das heißt wiederum, dass in Java jedes Objekt nicht nur als Hund, Button oder Snowboard behandelt werden kann, sondern auch als Object. Wenn Sie **`new Snowboard()`** sagen, erhalten Sie ein einzelnes Objekt auf dem Heap – ein Snowboard-Objekt –, aber dieses Snowboard umhüllt einen inneren Kern, der seinen Object-Teil (mit einem »c«) enthält.




Snowboard erbt Methoden von der Superklasse Object und fügt vier weitere hinzu.




Hier ist nur EIN Objekt auf dem Heap. Ein Snowboard-Objekt. Aber es enthält gleichzeitig die Snowboard-Teile seiner selbst und die Object-Teile seiner selbst.

# »Polymorphie« heißt »viele Formen«.

## Sie können ein Snowboard als Snowboard oder als Object behandeln.

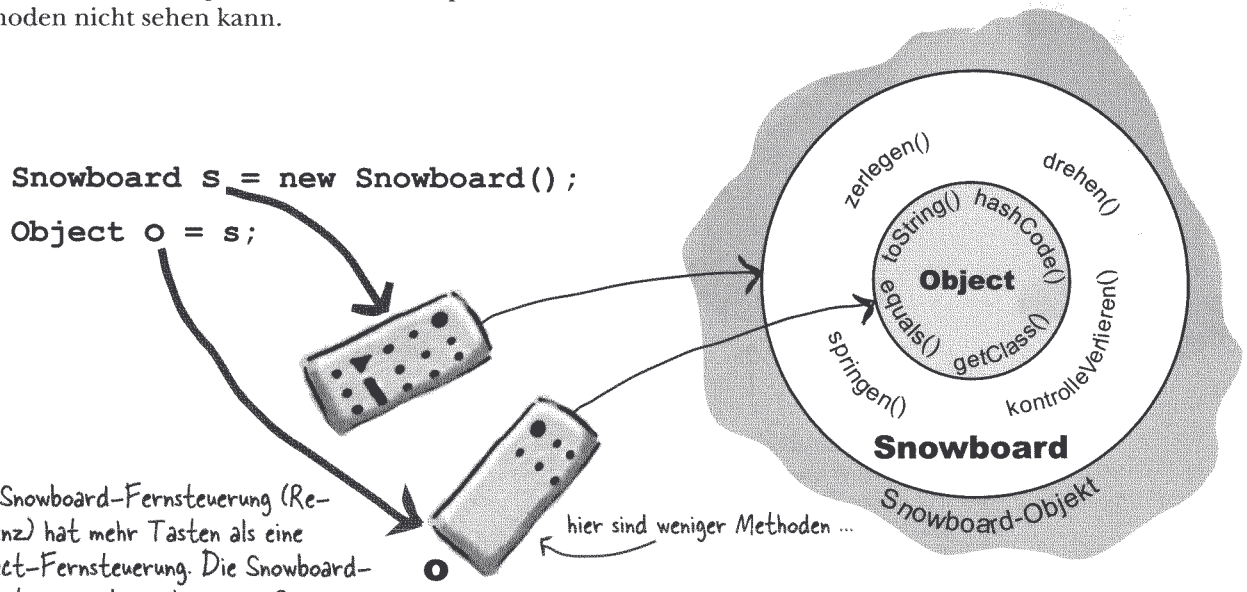
Wenn eine Referenz wie eine Fernsteuerung ist, erhält diese Fernsteuerung desto mehr Tasten, je weiter Sie sich im Vererbungsbaum nach unten bewegen. Eine Fernsteuerung (Referenz) des Typs Object hat nur ein paar Tasten – die Buttons für die veröffentlichten Methoden der Klasse Object. Aber eine Fernsteuerung des Typs Snowboard schließt alle Tasten der Klasse Object plus alle neuen Tasten (für neue Methoden) der Klasse Snowboard ein. Je spezifischer die Klasse ist, je mehr Tasten kann sie haben.

Natürlich stimmt das nicht immer. Eine Unterklasse muss nicht unbedingt neue Methoden hinzufügen, sondern kann auch einfach Methoden ihrer Superklasse überschreiben. Der springende Punkt ist, dass, selbst wenn das *Objekt* den Typ Snowboard hat, eine Object-*Referenz* auf das Snowboard-Objekt die Snowboard-spezifischen Methoden nicht sehen kann.

**Wenn Sie ein Objekt in eine ArrayList<Object> stecken, können Sie es nur als Object behandeln – egal welchen Typ es hatte, als Sie es hineinsteckten.**

**Wenn Sie eine Referenz von einer ArrayList<Object> erhalten, hat die Referenz immer den Typ Object.**

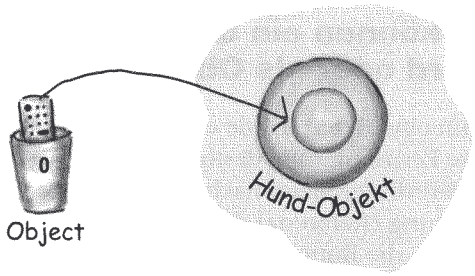
**Das heißt, dass Sie eine *Object*-Fernsteuerung erhalten.**




Die Snowboard-Fernsteuerung (Referenz) hat mehr Tasten als eine Object-Fernsteuerung. Die Snowboard-Fernsteuerung kann das ganze Snowboard-Dasein des Snowboard-Objekts sehen. Sie kann auf all seine Methoden zugreifen – auf die, die von Object geerbt wurden, und auf die, die aus der Klasse Snowboard stammen.

Die Object-Referenz kann nur die Object-Teile des Snowboard-Objekts sehen. Sie kann nur auf die Methoden der Klasse Object zugreifen. Sie hat weniger Tasten als die Snowboard-Fernsteuerung.

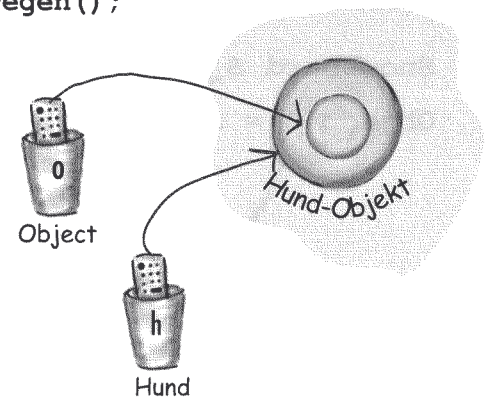
## Eine Objektreferenz auf Ihren *richtigen* Typ casten.




Eigentlich ist es immer noch ein Hund-*Objekt*, aber wenn Sie die Hund-spezifischen Methoden aufrufen möchten, brauchen Sie eine *Referenz* mit dem deklarierten Typ Hund. Wenn Sie *sicher** sind, dass das Objekt wirklich ein Hund ist, können Sie eine neue Hund-Referenz darauf erzeugen, indem Sie die Object-Referenz kopieren und diese Kopie mit dem Cast (Hund), einer Typumwandlung, zwingen, in eine Hund-Referenzvariable zu passen. Dann können Sie die neue *Hund*-Referenz verwenden, um *Hund*-Methoden aufzurufen.

```
Object o = al.get(index);
Hund h = (Hund) o;
h.bewegen();
```

Das Object wieder in den Hund zurückverwandeln, den wir dahinter wissen.



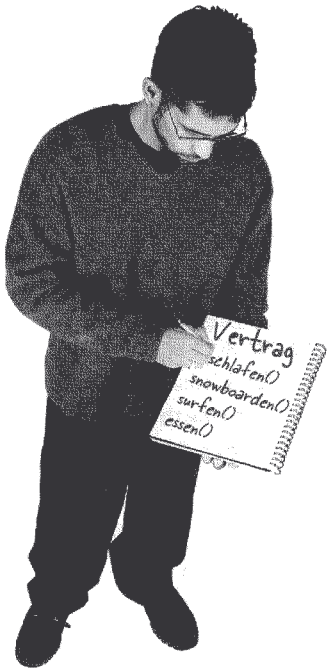

*Wenn Sie *nicht* sicher sind, dass es ein Hund ist, können Sie den **instanceof**-Operator verwenden, um das zu überprüfen. Denn sollten Sie sich irren, wenn Sie einen Cast ausführen, erhalten Sie zur Laufzeit eine ClassCastException und bleiben mit einem Kolbenfresser liegen.

```
if (o instanceof Hund) {
   Hund h = (Hund) o;
}
```

## Jetzt haben Sie gesehen, wie gründlich sich Java um die Methoden in der Klasse der Referenzvariablen kümmert.

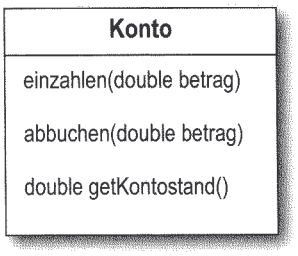
**Sie können eine Methode nur *dann* auf einem Objekt aufrufen, wenn die Klasse der Referenzvariablen diese Methode besitzt.**

**Betrachten Sie die öffentlichen Methoden in Ihrer Klasse als einen Vertrag – als ein Versprechen an die Außenwelt, was sie können.**




Wenn Sie eine Klasse schreiben, *veröffentlichen* Sie eigentlich immer einige der Methoden für Code außerhalb der Klasse. Eine Methode *veröffentlichen* heißt, dass Sie eine Methode *zugreifbar machen*, und zwar in der Regel, indem Sie sie mit public markieren.

Stellen Sie sich folgendes Szenario vor: Sie schreiben Code für ein Buchführungsprogramm. Eine für »Simons Surf-Shop« angepasste Anwendung. Als guter Wiederverwerter haben Sie eine Konto-Klasse gefunden, die Ihren Bedürfnissen perfekt zu entsprechen scheint, zumindest wenn man ihrer Dokumentation glauben kann. Jede Konto-Instanz repräsentiert das Konto eines einzelnen Kunden. Da stehen Sie also, kümmern sich um Ihre eigenen Angelegenheiten, rufen die Methoden *abbuchen()* und *einzahlen()* auf einem Konto-Objekt auf und stellen plötzlich fest, dass Sie den Kontostand für ein Konto abrufen müssen. Kein Problem – es gibt eine *getKontostand()*-Methode, die in Ordnung sein sollte.




Aber dann ... geht das ganze Ding zur Laufzeit hoch, als Sie versuchen, die Methode *getKontostand()* aufzurufen. Schmeißen Sie die Dokumentation weg, die Klasse hat die Methode nicht.

Autsch!

Aber das kann Ihnen gar nicht passieren, weil der Compiler jedes Mal, wenn Sie den Punktoperator auf eine Referenz (a.machZeug()) anwenden, auf den Typ der *Referenz* blickt (der Typ, als der ›a‹ deklariert wurde) und diese Klasse prüft, um sicherzustellen, dass die Klasse die Methode hat und dass diese Methode tatsächlich die Argumente nimmt, die Sie übergeben, und den Typ Wert zurückliefert, den Sie zurückerwarten.

**Denken Sie einfach daran, dass der Compiler die Klasse der *Referenzvariablen* prüft, nicht die Klasse des *Objekts*, das sich tatsächlich am Ende der Referenz befindet.**

# Was ist, wenn man einen Vertrag ändern muss?

Gut, nehmen wir mal an, Sie sind ein Hund. Ihre Hund-Klasse ist nicht der *einzige* Vertrag, der definiert, wer Sie sind. Denken Sie daran, dass Sie die zugreifbaren (was in der Regel heißt, *öffentlichen*) Methoden all Ihrer Superklassen erben.

Es stimmt, dass Ihre Hund-Klasse einen Vertrag definiert.

Aber nicht Ihren *ganzen* Vertrag.

**Alles in der Klasse *Hundeartig* ist Teil Ihres Vertrags.**

**Alles in der Klasse *Tier* ist Teil Ihres Vertrags.**

**Alles in der Klasse *Object* ist Teil Ihres Vertrags.**

Gemäß dem IS-A-Test *sind* Sie all diese Dinge – Hundeartig, Tier und Object.

Aber was ist, wenn die Person, die Ihre Klasse entwarf, eine Tier-Simulation im Sinn hatte und Sie (die Klasse Hund) jetzt für eine populärwissenschaftliche Einführung zu Tier-Objekten verwenden will?

Wahrscheinlich ist das in Ordnung. Wahrscheinlich können Sie für diesen Zweck wiederverwendet werden.

Aber was ist, wenn er Sie später für ein Tierhandlungsprogramm wiederverwenden will? *Sie haben kein **Haustier**verhalten.* Ein Haustier braucht Methoden wie *seiFreundlich()* und *spielen()*.

Gut, nehmen wir jetzt an, Sie sind der Programmierer der Klasse Hund. Kein Problem, oder? Sie fügen der Klasse Hund einfach ein paar zusätzliche Methoden hinzu. Wenn Sie Methoden *hinzufügen*, machen Sie keinen Code von anderen kaputt, weil Sie die *bestehenden* Methoden nicht anrühren, die vielleicht bereits jemand auf Hund-Objekten aufruft.

Können Sie die Nachteile dieses Ansatzes erkennen (der Klasse Hund Haustier-Methoden hinzuzufügen)?

Denken Sie darüber nach, was **SIE** tun würden, wenn **SIE** der Programmierer der Klasse Hund wären und Hund so modifizieren müssten, dass sie auch Haustier-Dinge tun könnte. Wir wissen, dass es möglich wäre, der Klasse Hund einfach neues Haustier-Verhalten (Methoden) hinzuzufügen, ohne dass dadurch der Code anderer zerbrochen würde.

Aber ... das ist ein Tierhandlungsprogramm. Es hat nicht nur Hunde! Und was ist, wenn jemand Ihre Hund-Klasse für wilde Hunde verwenden will? Was, glauben Sie, sind Ihre Optionen? Machen Sie sich einmal keine Gedanken darüber, wie Java mit den Dingen umgeht, und überlegen Sie sich einfach, wie Sie das Problem lösen würden, einige Ihrer Tier-Klassen so zu modifizieren, dass sie Haustierverhalten einschließen.

Machen Sie eine Pause und denken Sie darüber nach, **bevor Sie auf die nächste Seite blicken,** auf der wir beginnen, *alles* zu offenbaren.

(und so die ganze Übung nutzlos machen und sich der Einen Großen Chance berauben, ein paar Gehirn-Kalorien zu verbrennen)

# Sehen wir uns ein paar der Entwurfsoptionen zur Wiederverwendung einiger unserer vorhandenen Tier-Klassen in einem Tierhandlungsprogramm an.

Auf den nächsten paar Seiten werden wir einige Möglichkeiten durchgehen. Dabei kümmern wir uns nicht darum, ob Java das, was wir uns ausdenken, tatsächlich *kann*. Diese Brücke schlagen wir erst, wenn wir eine gute Vorstellung von den verschiedenen Vor- und Nachteilen haben.

## ① Option eins

Wir wählen den einfachen Weg und stecken einige der Haustier-Methoden in die Klasse Tier.

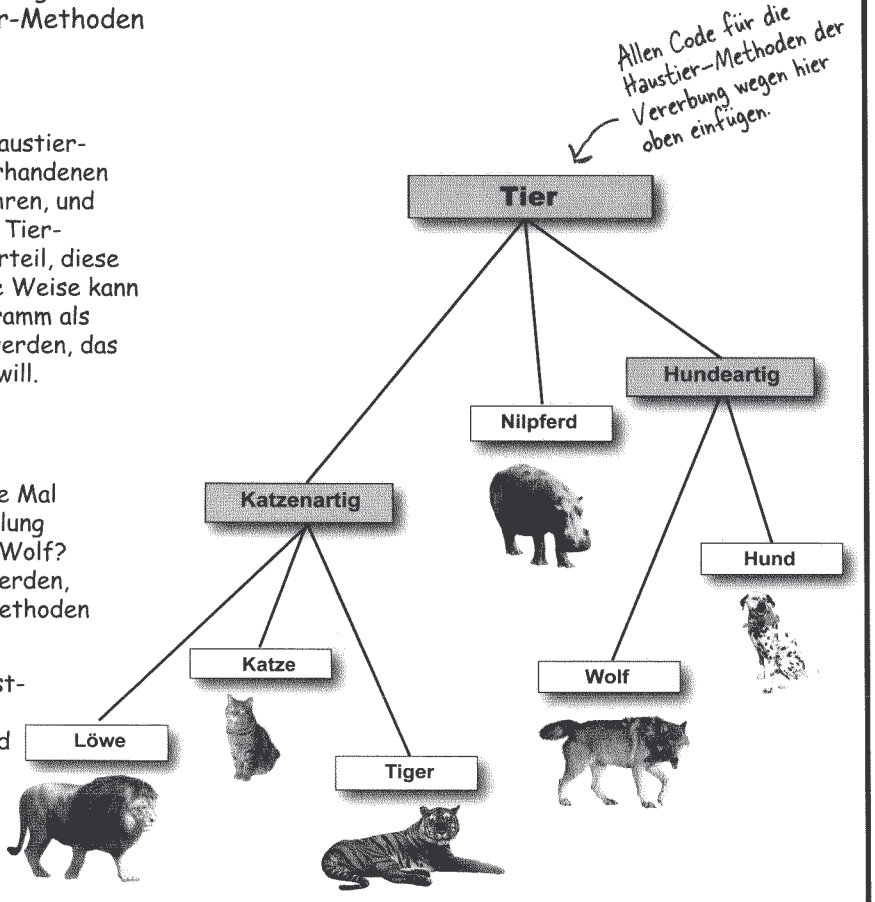
### Pro:

Alle Tiere erben direkt das Haustier-Verhalten. Wir müssen die vorhandenen Tier-Unterklassen nicht anrühren, und alle in Zukunft geschriebenen Tier-Unterklassen genießen den Vorteil, diese Methoden zu erben. Auf diese Weise kann die Klasse Tier in jedem Programm als polymorpher Typ verwendet werden, das Tier als Haustiere behandeln will.

### Kontra:

Tja, wann haben Sie das letzte Mal ein Nilpferd in einer Tierhandlung gesehen? Einen Löwen? Einen Wolf? Könnte vielleicht gefährlich werden, Nicht-Haustieren Haustier-Methoden zu spendieren.

Außerdem MÜSSEN wir höchstwahrscheinlich Hand an die Haustier-Klassen wie Hund und Katze legen, weil (zumindest bei uns zu Hause) Hunde und Katzen dazu neigen, Haustier-Verhalten SEHR unterschiedlich zu implementieren.

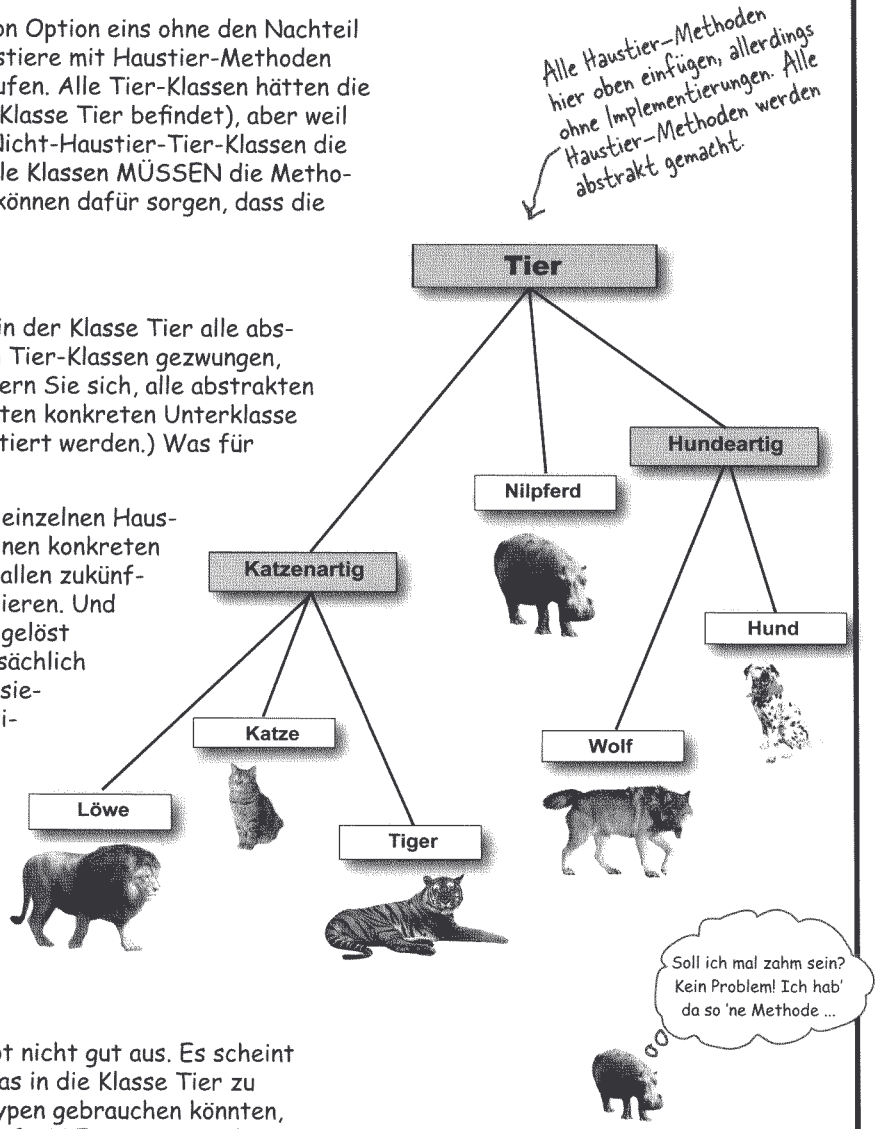


## ② Option zwei

Wir beginnen mit Option eins und stecken die Haustier-Methoden in die Klasse Tier, machen die Methoden aber abstrakt und zwingen die Tier-Unterklassen so, sie zu überschreiben.

### Pro:

Das würde uns alle Vorteile von Option eins ohne den Nachteil verschaffen, dass Nicht-Haustiere mit Haustier-Methoden (wie seiFreundlich()) herumlaufen. Alle Tier-Klassen hätten die Methode (weil sie sich in der Klasse Tier befindet), aber weil sie abstrakt ist, würden die Nicht-Haustier-Tier-Klassen die Funktionalität nicht erben. Alle Klassen MÜSSEN die Methoden überschreiben, aber Sie können dafür sorgen, dass die Methoden »nichts tun«.




### Kontra:

Weil die Haustier-Methoden in der Klasse Tier alle abstrakt sind, sind die konkreten Tier-Klassen gezwungen, alle zu implementieren. (Erinnern Sie sich, alle abstrakten Klassen MÜSSEN von der ersten konkreten Unterklasse im Vererbungsbaum implementiert werden.) Was für eine Zeitverschwendung!

Sie müssen da sitzen und alle einzelnen Haustier-Methoden in jeder einzelnen konkreten Nicht-Haustier-Klasse und in allen zukünftigen Unterklassen implementieren. Und auch wenn damit das Problem gelöst ist, dass Nicht-Haustiere tatsächlich Haustier-Dinge TUN (was passieren würde, wenn sie die Funktionalität von der Klasse Tier erben würden), bleibt der Vertrag schlecht. Jede *Nicht*-Haustier-Klasse würde der Welt verkünden, dass auch sie die Haustier-Methoden hat, obwohl diese eigentlich nichts TUN, wenn sie aufgerufen werden.

Dieser Ansatz sieht überhaupt nicht gut aus. Es scheint einfach falsch zu sein, alles das in die Klasse Tier zu stecken, was mehrere Tier-Typen gebrauchen könnten, wenn es NICHT WIRKLICH auf ALLE Tier-Unterklassen anwendbar ist.

## ③ Option drei

Die Haustier-Methoden NUR in die Klassen stecken, in die sie gehören.

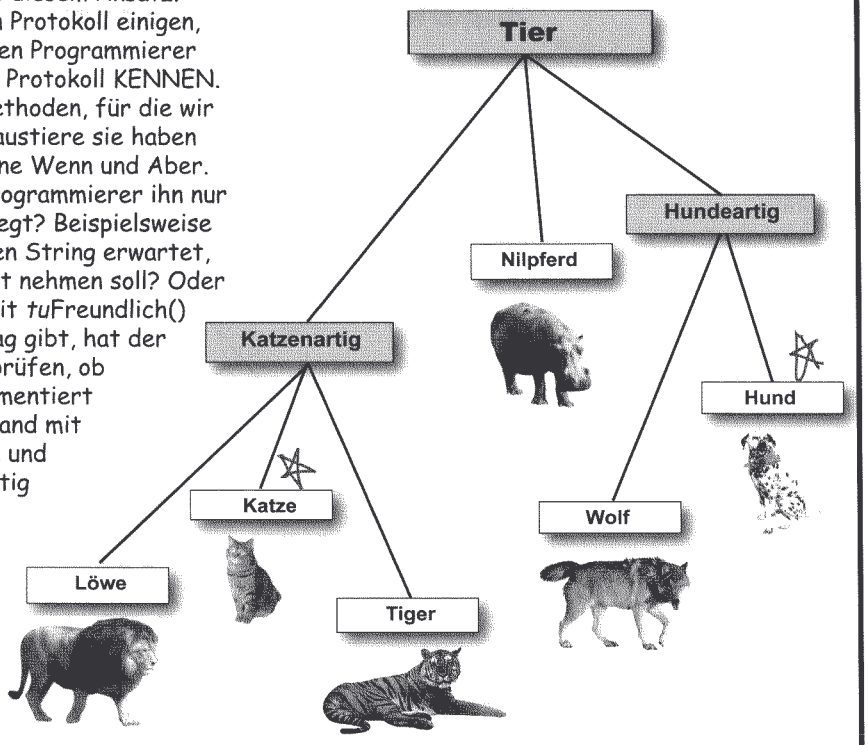
### Pro:

Wir müssen uns keine Gedanken mehr über Nilpferde machen, die einen schwanzwedelnd an der Tür begrüßen oder einem das Gesicht ablecken. Die Methoden gibt es dort und NUR dort, wo sie hingehören. Hunde können die Methoden implementieren, und Katzen können sie implementieren, und sonst muss sich niemand um sie kümmern.

Die Haustier-Methoden nicht in Tier, sondern NUR in die Tier-Klassen stecken, die Haustiere sein können.

### Kontra:

Es gibt zwei große Probleme bei diesem Ansatz. Erstens müssen Sie sich auf ein Protokoll einigen, und alle heutigen und zukünftigen Programmierer von Tier-Klassen müssen dieses Protokoll KENNEN. Mit Protokoll meinen wir die Methoden, für die wir entschieden haben, dass alle Haustiere sie haben sollen. Ein Haustier-Vertrag ohne Wenn und Aber. Aber was ist, wenn einer der Programmierer ihn nur ein kleines bisschen falsch auslegt? Beispielsweise eine Methode schreibt, die einen String erwartet, obwohl diese eigentlich einen int nehmen soll? Oder *sei*Freundlich() aus Versehen mit *tu*Freundlich() übersetzt? Da es keinen Vertrag gibt, hat der Compiler keine Möglichkeit zu prüfen, ob Sie die Methoden richtig implementiert haben. Ganz schnell könnte jemand mit Haustier-Tier-Klassen arbeiten und feststellen, dass nicht alle richtig funktionieren.

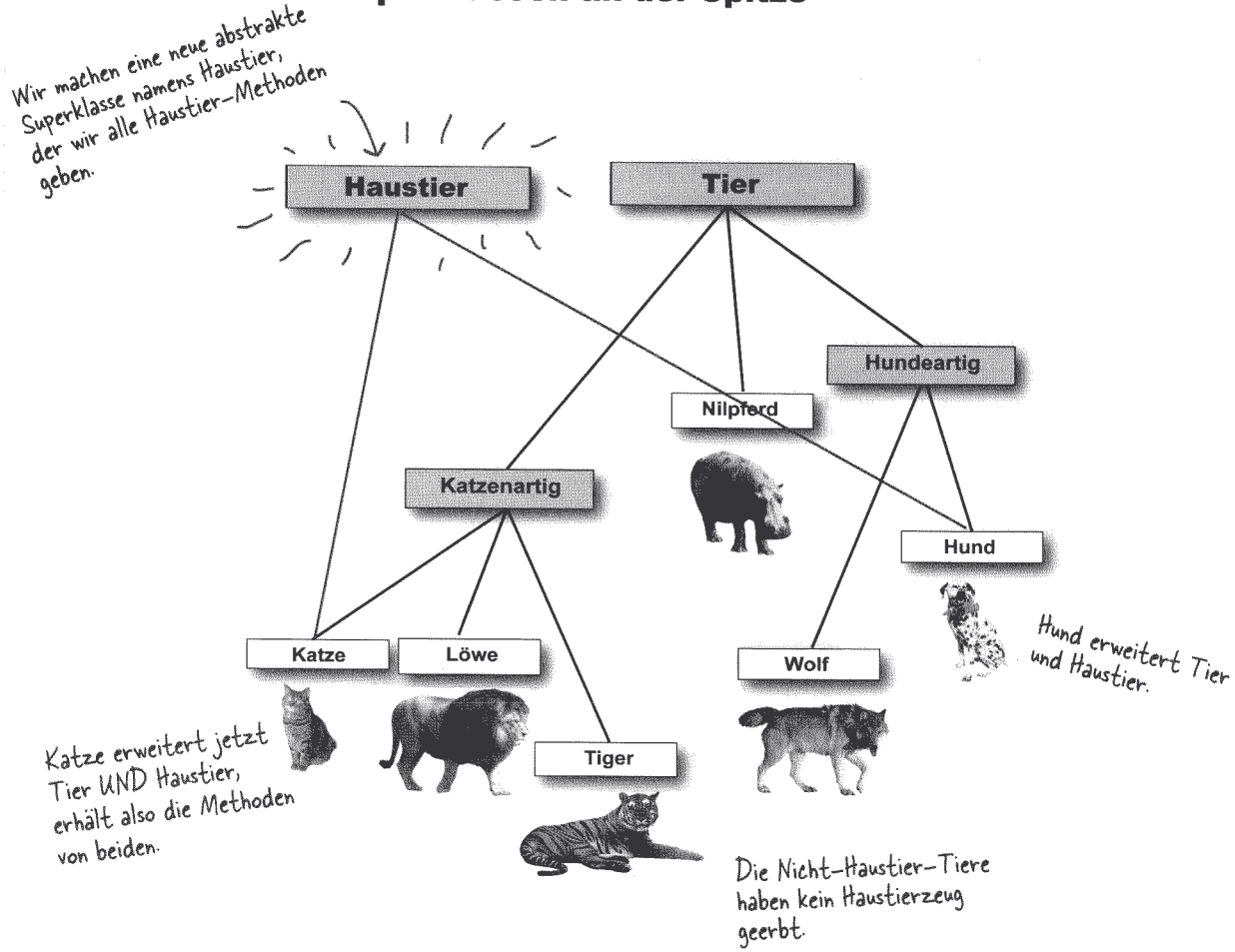



Und zweitens können Sie für die Haustier-Methoden keine Polymorphie verwenden. Jede Klasse, die Haustier-Verhalten verwendet, müsste jede einzelne Klasse kennen! Anders gesagt, Sie können Tier nicht mehr als polymorphen Typ verwenden, weil der Compiler Ihnen nicht erlaubt, Haustier-Methoden auf einer Tier-Referenz aufzurufen (selbst wenn es eine Hund-Referenz ist), weil die Klasse Tier die Methode nicht hat.

## Was wir WIRKLICH brauchen, ist also:

- Einen Weg, das Haustier-Verhalten **nur** in den Haustier-Klassen zu haben.
- Einen Weg, der sichert, dass alle Haustier-Klassen die gleichen Methoden definieren (den gleichen Namen, die gleichen Argumente, den gleichen Rückgabetyp usw.), ohne dass wir die Daumen drücken und hoffen müssen, dass alle Programmierer das richtig hinbekommen.
- Einen Weg, der die Vorteile der Polymorphie nutzen kann, damit die Haustier-Methoden aller Haustiere aufgerufen werden können, ohne Argumente, Rückgabetypen und Arrays für alle einzelnen Haustier-Klassen zu verwenden.

### Es scheint, als bräuchten wir ZWEI Superklassen an der Spitze



Es gibt nur ein Problem mit diesem »Zwei-Superklassen«-Ansatz!

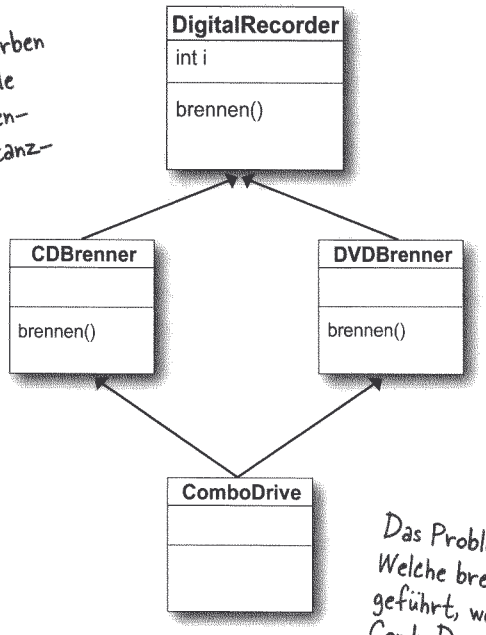
## Es wird »Mehrfachvererbung« genannt und kann eine richtig üble Angelegenheit sein.

Das heißt, wenn es in Java möglich wäre, so was zu tun.

Aber das ist es nicht, weil Mehrfachvererbung ein Problem birgt, das als »The Deadly Diamond of Death« bezeichnet wird (also als »Tödlicher Diamant des Todes« bzw. »Tödliches Todeskaro«).

### »The Deadly Diamond of Death«

CDBrenner und DVDBrenner erben beide von DigitalRecorder, beide überschreiben die Methode brennen(), und beide erben die Instanzvariable »i«.




Stellen Sie sich vor, die Instanzvariable »i« wird von CDBrenner und DVDBrenner verwendet und jeweils auf unterschiedliche Werte gesetzt. Was passiert, wenn ComboDrive beide Werte von »i« braucht?

Das Problem bei Mehrfachvererbung. Welche brennen()-Methode wird ausgeführt, wenn man brennen() auf dem ComboDrive aufruft?

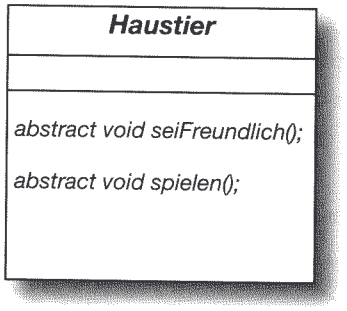
In einer Sprache, die den Deadly Diamond of Death zulässt, kann es zu einigen hässlichen Komplexitäten kommen, weil man spezielle Regeln braucht, um mögliche Mehrdeutigkeiten aufzulösen. Und zusätzliche Regeln bedeuten zusätzliche Arbeit. Sie müssen diese Regeln *lernen*, und dann müssen Sie auch noch nach diesen »besonderen Fällen« Ausschau halten. Java soll *einfach* sein, konsistente Regeln haben, die nicht bei gewissen Konstellationen hochgehen. Deswegen schützt Sie Java (anders als C++) davor, sich über das Todeskaro Sorgen machen zu müssen. Aber das führt uns wieder zu unserem Ausgangsproblem zurück! *Wie machen wir das mit der Tier/Haustier-Geschichte?*

# Die Rettung: Interfaces!

Java bietet Ihnen eine Lösung. Eine *Schnittstelle.* Keine *GUI*-Schnittstelle, nicht in der allgemeinen Verwendung des *Worts* Schnittstelle, beispielsweise in »Das ist die öffentliche Schnittstelle der API der Button-Klasse«, sondern das Java-*Schlüsselwort* **`interface`**. Wir werden deswegen von jetzt an solche Schnittstellen auch als »Interface« bezeichnen.

Ein Java-Interface löst Ihr Mehrfachvererbungsproblem, indem es Ihnen die polymorphen *Vorteile* von Mehrfachvererbung ohne den Schmerz und das Leiden durch den Tödlichen Diamanten des Todes bietet.

Die Art, auf die Interfaces mit dem Tödlichen Diamanten des Todes umgehen, ist überraschend einfach: ***mach alle Methoden abstrakt!*** Auf diese Weise ***muss*** die Unterklasse die Methoden implementieren (denken Sie daran, abstrakte Methoden *müssen* von der ersten konkreten Unterklasse implementiert werden). Und zur Laufzeit muss sich die JVM nicht fragen, *welche* der beiden geerbten Methoden sie wohl aufrufen soll.




**Ein Java-Interface ist wie eine 100% reine abstrakte Klasse.**

Alle Methoden in einem Interface sind abstrakt. Also MUSS jede Klasse, für die Ist-Ein Haustier gilt, die Methoden von Haustier implementieren (d.h. überschreiben).

**Um ein Interface zu DEFINIEREN:**

```
public interface Haustier {...}
```

Verwenden Sie statt des Schlüsselworts »class« das Schlüsselwort »interface«.

**Um ein Interface zu IMPLEMENTIEREN:**

```
public class Hund extends Hundeartig implements Haustier {...}
```

Verwenden Sie das Schlüsselwort »implements« gefolgt vom Namen des Interface. Beachten Sie, dass Sie, wenn Sie ein Interface implementieren, immer noch eine Klasse erweitern können.

# Ein Haustier-Interface erstellen und implementieren

Interface-Methoden sind implizit öffentlich und abstrakt, die Eingabe von »public« und »abstract« ist also optional (eigentlich wird es sogar als schlechter Stil betrachtet, wenn man die Wörter eingibt, aber wir haben das hier gemacht, um sie zu betonen und weil wir uns niemals zu Sklaven von Moden machen lassen ...).

Hier sagen Sie »interface« statt »class«.

```
public interface Haustier {

   public abstract void seiFreundlich();

   public abstract void spielen();

}
```

Alle Interface-Methoden sind abstrakt, MÜSSEN also mit Semikola enden. Denken Sie daran: Sie haben keinen Body!

Hund Ist-Ein Tier und Hund Ist-Ein Haustier

Sie sagen »implements« gefolgt vom Namen des Interface.

```
public class Hund extends Hundeartig implements Haustier {

   public void seiFreundlich() {...}

   public void spielen() {..}

   public void bewegen() {...}

   public void essen() {...}

}
```

Sie SAGTEN, Sie seien ein Haustier, also MÜSSEN Sie die Haustier-Methoden implementieren. Das ist Ihr Vertrag. Achten Sie auf die geschweiften Klammern an Stelle der Semikola.

Das sind bloß gewöhnliche überschreibende Methoden.

## Es gibt keine Dummen Fragen

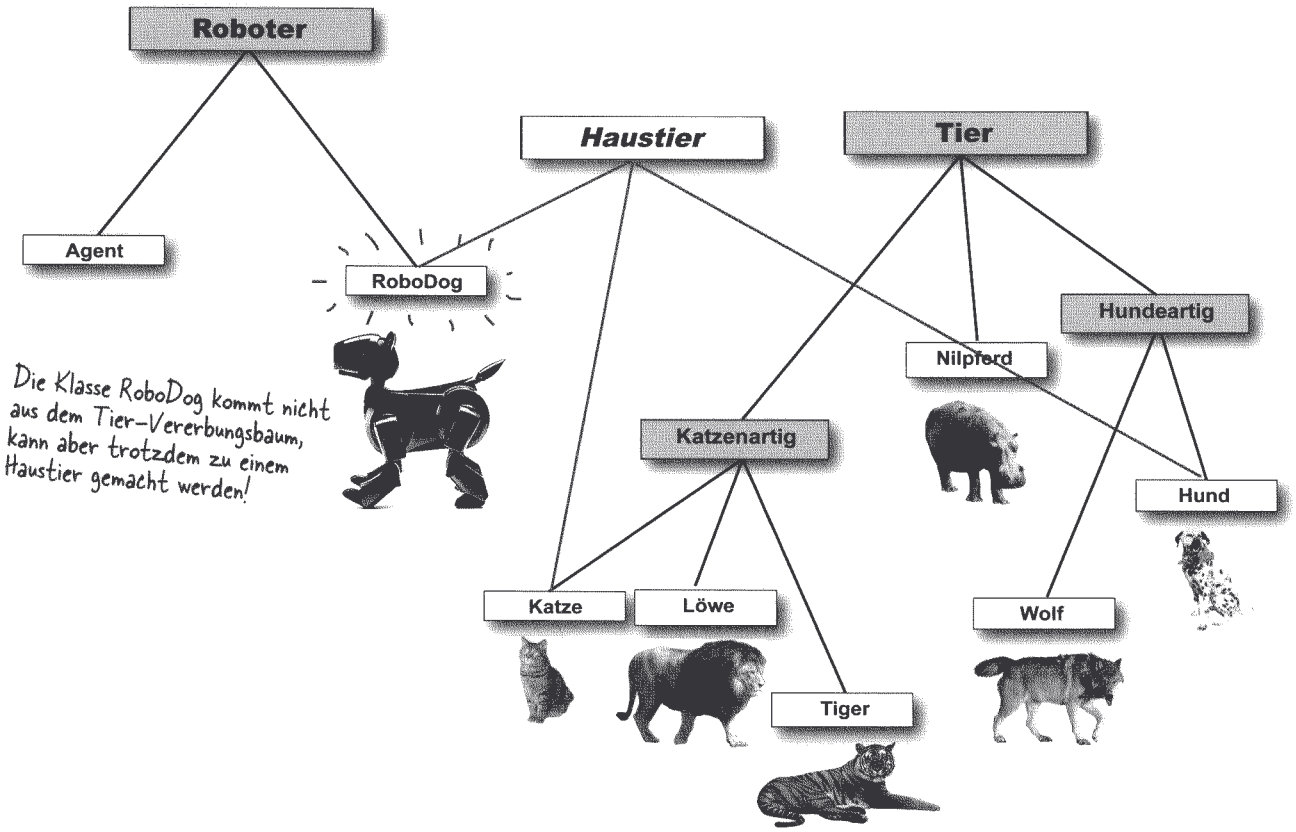
**F: Moment mal ... Schnittstellen, äh, Interfaces, geben Ihnen keine Mehrfachvererbung, weil Sie keinen Implementierungscode in sie stecken können. Aber wenn alle Methoden abstrakt sind, was bringen sie Ihnen dann eigentlich?**

A: Polymorphie, Polymorphie, Polymorphie. Interfaces sind das ultimative Werkzeug für Flexibilität, weil Sie Interfaces an Stelle konkreter Unterklassen (oder sogar abstrakter Superklassen) als Argument- und Rückgabetypen verwenden können. Dann können Sie alles überreichen, das dieses Interface implementiert. Und denken Sie daran – wenn Sie Interfaces haben, muss eine Klasse nicht aus einem einzigen Vererbungsbaum kommen. Eine Klasse kann eine Klasse erweitern und ein Interface implementieren. Und eine andere Klasse, die dasselbe Interface implementiert, kann trotzdem aus einem ganz anderen Vererbungsbaum kommen! Auf diese Weise können Sie ein Objekt in Abhängigkeit von der Rolle verarbeiten, die es spielt, und nicht nur auf Basis der Klasse, von der es instantiiert wurde.

Wenn Sie Ihren Code so geschrieben haben, dass er Interfaces verwendet, müssten Sie eigentlich niemandem eine zu erweiternde Superklasse geben. Sie könnten ihnen auch einfach nur das Interface geben und sagen: »Hier, mir ist egal, aus was für einer Vererbungsstruktur ihr kommt, implementiert einfach dieses Interface, und ihr könnt gleich loslegen.«

Es stellt sich heraus, dass der Umstand, dass Sie keinen Implementierungscode angeben können, bei den meisten guten Designs kein Problem ist. Bei den meisten Interface-Methoden würde es einfach keinen Sinn ergeben, sie auf allgemeine Weise zu implementieren. Mit anderen Worten: Die meisten Interface-Methoden müssten auch dann überschrieben werden, wenn die Methoden nicht *gezwungenermaßen* abstrakt wären.

## Klassen aus *unterschiedlichen* Vererbungsbäumen können das *gleiche* Interface implementieren.



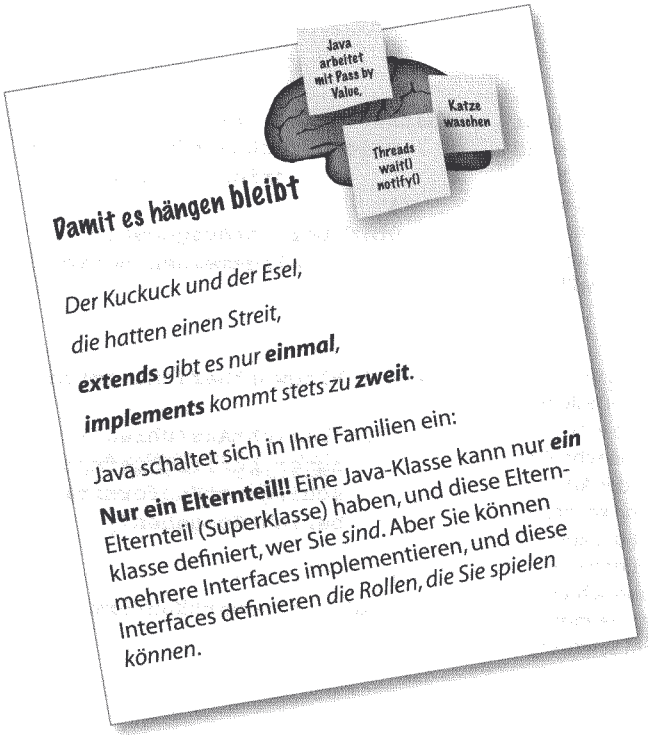

Wenn Sie eine *Klasse* als polymorphen Typ verwenden (wie bei einem Array vom Typ Tier oder einer Methode, die ein Argument vom Typ Hundeartig nimmt), müssen die Objekte, die Sie in diesen Typ stecken, aus dem gleichen Vererbungsbaum kommen. Und die Objekte können nicht von einer beliebigen Stelle des Vererbungsbaums kommen, sondern müssen Instanzen einer Klasse sein, die eine Unterklasse des polymorphen Typs ist. Ein Argument des Typs Hundeartig akzeptiert einen Wolf und einen Hund, aber keine Katze und kein Nilpferd.

Aber wenn Sie ein ***Interface*** als polymorphen Typ verwenden (ein Array von Haustieren beispielsweise), können die Objekte von einer *beliebigen* Stelle im Vererbungsbaum kommen. Die einzige Forderung ist, dass die Objekte Instanzen einer Klasse sind, die das Interface *implementiert*. Dass Klassen aus unterschiedlichen Vererbungsbäumen ein gemeinsames Interface implementieren können, ist für die Java-API von entscheidender Bedeutung. Möchten Sie, dass ein Objekt dazu in der Lage ist, seinen Zustand in einer Datei zu speichern? Implementieren Sie einfach das Interface Serializable. Brauchen Sie Objekte, die ihre Methoden in separaten Threads ausführen können? Implementieren Sie Runnable. Verstehen Sie? Mehr Informationen zu Serializable und Runnable erhalten Sie in späteren Kapiteln. Merken Sie sich jetzt einfach nur, dass Klassen von beliebigen Stellen im Vererbungsbaum diese Interfaces eventuell implementieren müssen. Fast *alle* Klassen möchten eventuell Serializable oder Runnable sein.

### Noch besser ist, dass eine Klasse *mehrere* Interfaces implementieren kann!

Ein Hund-Objekt Ist-Ein Hundeartig und Ist-Ein Tier und Ist-Ein Object – alles durch Vererbung. Aber Hund Ist-Ein Haustier durch Interface-Implementierung, und der Hund könnte auch noch weitere Interfaces implementieren. Sie könnten Folgendes sagen:

```
public class Hund extends Tier implements
Haustier, Serializable, Paintable { ... }
```




## Woher wissen Sie, ob Sie eine Klasse, eine Unterklasse, eine abstrakte Klasse oder ein Interface anlegen sollten?

- Machen Sie eine Klasse, die (außer Object) nichts implementiert, wenn Ihre neue Klasse keinen IS-A-Test für einen anderen Typ besteht.
- Machen Sie nur dann eine Unterklasse (erweitern Sie eine Klasse), wenn Sie eine ***spezifischere*** Version einer Klasse machen müssen, weil Sie Verhalten überschreiben oder neue Verhalten hinzufügen müssen.
- Machen Sie eine abstrakte Klasse, wenn Sie eine ***Schablone*** für eine Gruppe von Unterklassen machen möchten und wenn Sie zumindest *etwas* Implementierungscode haben, den alle Klassen nutzen können. Machen Sie die Klasse abstrakt, wenn Sie sicherstellen wollen, dass niemand Objekte dieses Typs erzeugen kann.
- Verwenden Sie ein Interface, wenn Sie eine ***Rolle*** definieren wollen, die andere Klassen unabhängig davon, an welcher Stelle sie sich im Vererbungsbaum befinden, spielen können.

## Die Superklassenversion einer Methode aufrufen

**F: Was ist, wenn man eine konkrete Unterklasse erstellt und eine Methode überschreiben muss und dabei das Verhalten der Superklassenversion dieser Methode verwenden will? Anders gesagt, was macht man, wenn man die Methode nicht durch eine Überschreibung *ersetzen*, sondern ihr nur zusätzlichen spezifischen Code *hinzufügen* will?**

A: Ahhh ... denken Sie an die Bedeutung des Worts »extends« (erweitert). Ein Bereich guten OO-Designs befasst sich damit, wie man konkreten Code entwirft, der überschrieben werden *soll*. Das heißt, Sie schreiben Methodencode in einer abstrakten Klasse, der Arbeit macht, die allgemein genug ist, um typische konkrete Implementierungen zu unterstützen. Aber der konkrete Code reicht nicht aus, um die *ganze* unterklassenspezifische Arbeit zu erledigen. Also überschreibt die Unterklasse die Methode und *erweitert* sie, indem sie den restlichen Code hinzufügt. Mit dem Schlüsselwort super können Sie die Superklassenversion einer überschriebenen Methode aus der Unterklasse heraus aufrufen.

Superklassenversion der Methode, die wichtigen Kram erledigt, den die Unterklassen nutzen könnten.

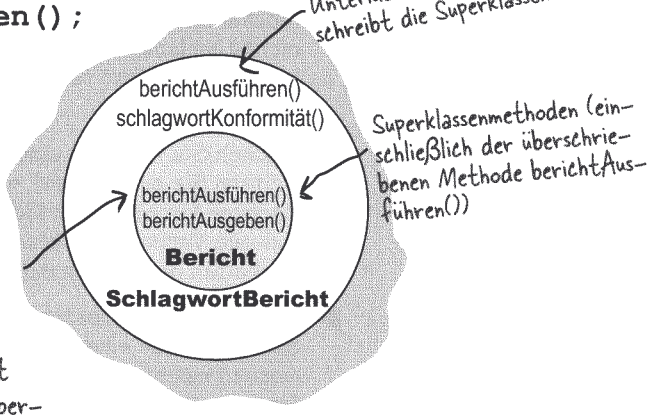
```
abstract class Bericht {
    void berichtAusführen() {
        // Bericht einrichten
    }
    void berichtAusgeben() {
        // allgemeinen Bericht ausgeben
    }
}

class SchlagwortBericht extends Bericht {

    void berichtAusführen() {
        super.berichtAusführen();
        schlagwortKonformität();
        berichtAusgeben();

    }
    void schlagwortKonformität() {...}
}
```

Die Superklassenversion aufrufen und dann zurückkehren, um unterklassenspezifischen Kram zu erledigen.

Wenn Methodencode in der Unterklasse SchlagwortBericht

```
super.berichtAusführen();
```

sagt, wird die berichtAusführen()-Methode in der Superklasse Bericht ausgeführt.




**super.berichtAusführen();**

Eine Referenz auf eine Instanz der Unterklasse (SchlagwortBericht) ruft immer die Unterklassenversion einer überschriebenen Methode auf. Das ist Polymorphie. Aber der Unterklassencode kann super.berichtAusführen() verwenden, um die Superklassenversion aufzurufen.

Das Schlüsselwort super ist tatsächlich eine Referenz auf den Superklassenteil eines Objekts. Wenn der Unterklassencode wie in super.berichtAusführen() super verwendet, wird die Superklassenversion der Methode ausgeführt.

## Punkt für Punkt

- Wenn Sie verhindern möchten, dass eine Klasse instantiiert wird (d.h. nicht wollen, dass jemand ein Objekt dieser Klasse erstellt), markieren Sie die Klasse mit dem Schlüsselwort **`abstract`**.
- Eine abstrakte Klasse kann abstrakte und nicht-abstrakte Methoden haben.
- Wenn eine Klasse nur eine *einzige* abstrakte Methode hat, muss sie als abstract markiert sein.
- Eine abstrakte Methode hat keinen Body, und die Deklaration endet mit einem Semikolon (es gibt keine geschweiften Klammern).
- Alle abstrakten Methoden müssen in der ersten konkreten Unterklasse im Vererbungsbaum implementiert werden.
- In Java ist jede Klasse entweder eine direkte oder eine indirekte Unterklasse der Klasse **Object** (java.lang.Object).
- Methoden können mit Object als Argument- und/oder Rückgabetyp deklariert werden.
- Unabhängig davon, welchen Typ das eigentliche *Objekt* hat, können Sie auf einem Objekt *nur* Methoden aufrufen, die in der Klasse (oder dem Interface) deklariert sind, die (das) als Typ der *Referenzvariablen* verwendet wird. Eine Referenzvariable des Typs Object kann also nur verwendet werden, um Methoden aufzurufen, die in der Klasse Object definiert sind – egal welchen Typ das Objekt hat, das die Referenz referenziert.
- Eine Referenzvariable des Typs Object kann einer Referenzvariablen eines anderen Typs nur mit einem *Cast* (einer Typumwandlung) zugewiesen werden. Eine Typumwandlung kann verwendet werden, um eine Referenzvariable eines Typs einer Referenzvariablen eines Untertyps zuzuweisen. Aber die Typumwandlung schlägt zur Laufzeit fehl, wenn der Typ des Objekts auf dem Heap NICHT mit dem Typ der Typumwandlung kompatibel ist.

  Beispiel: `Hund h = (Hund) x.getObject(einHund);`
- Aus einer ArrayList<Object> kommen alle Objekte mit dem Typ Object heraus (d.h., sie können nur mit einer Referenzvariablen vom Typ Object referenziert werden, sofern Sie keine *Typumwandlung* verwenden).
- Die Mehrfachvererbung ist in Java auf Grund der mit dem »Deadly Diamond of Death« verbundenen Probleme nicht zulässig. Das bedeutet, dass Sie nur eine Klasse erweitern können (d.h., Sie können nur eine unmittelbare Superklasse haben).
- Ein Interface ist wie eine 100% reine abstrakte Klasse. Es definiert *nur* abstrakte Methoden.
- Ein Interface erstellen Sie, indem Sie das Schlüsselwort **`interface`** statt des Schlüsselworts **`class`** verwenden.
- Sie implementieren ein Interface, indem Sie das Schlüsselwort **`implements`** verwenden.

  Beispiel: `Hund implements Haustier`
- Ihre Klasse kann mehrere Interfaces implementieren.
- Eine Klasse, die ein Interface implementiert, *muss* alle Methoden aus dem Interface implementieren, da ***alle Interface-Methoden implizit public und abstract sind***.
- Um aus einer Unterklasse, die eine Methode überschreibt, die Superklassenversion aufzurufen, verwenden Sie das Schlüsselwort **super**.

  Beispiel: `super.berichtAusführen();`

**F: Aber eine Sache finde ich immer noch seltsam ... Sie haben immer noch nicht erklärt, warum ArrayList<Hund> eine Hund-Referenz zurückliefert, die nicht gecastet werden muss, obwohl die Klasse ArrayList in ihren Methoden Object und nicht Hund (oder DotCom oder irgendetwas anderes) verwendet. Was für ein besonderer Trick wird verwendet, wenn man ArrayList<Hund> sagt?**

A: Sie haben schon Recht, wenn Sie von einem besonderen Trick sprechen. Es ist tatsächlich ein besonderer Trick, dass ArrayList<Hund> Hunde zurückliefert, ohne dass Sie eine Typumwandlung durchführen müssen, obwohl es scheint, als wüssten die Methoden von ArrayList nichts über Hund oder irgendeinen anderen Typ außer Object.

Die kurze Antwort ist, dass *der Compiler den Cast für Sie einsetzt!* Wenn Sie ArrayList<Hund> sagen, gibt es keine besondere Klasse, die Methoden hat, die Hund-Objekte nehmen und zurückliefern. Stattdessen signalisiert <Hund> dem Compiler, dass Sie von ihm verlangen, dass er Sie NUR Hund-Objekte in die Liste stecken lassen soll und Sie davon abhalten soll zu versuchen, der Liste ein Objekt eines anderen Typs hinzuzufügen. Und da der Compiler verhindert, dass Sie der ArrayList irgendetwas anderes als Hunde hinzufügen, weiß der Compiler auch, dass es sicher ist, alles, was aus dieser ArrayList kommt, auf eine Hund-Referenz zu casten. Mit anderen Worten: Wenn Sie ArrayList<Hund> verwenden, erspart Ihnen das, den Hund umzuwandeln, den Sie zurückbekommen. Aber es ist noch viel wichtiger als das ... denn denken Sie daran, eine Typumwandlung kann zur Laufzeit fehlschlagen. Ist es Ihnen nicht lieber, dass Ihre Fehler zur Kompilierzeit auftreten und nicht, wenn Ihr Benutzer das Programm im Produktivbetrieb einsetzt?

Aber dazu gibt es noch viel mehr zu erzählen, und das werden wir uns alles im Collections-Kapitel ansehen.

Hier ist Ihre Chance, Ihre künstlerischen Fähigkeiten zu demonstrieren. Auf der linken Seite finden Sie mehrere Sätze mit Klassen- und Interface-Deklarationen. Ihre Aufgabe ist es, auf der rechten Seite die entsprechenden Klassendiagramme zu zeichnen. Das erste haben wir für Sie schon angefertigt. Verwenden Sie für »implements« eine gestrichelte und für »extends« eine durchgezogene Linie.

**Voraussetzung:**

**1)**
```
public interface Foo { }
public class Bar implements Foo { }
```

**2)**
```
public interface Vein { }
public abstract class Vaus implements Vein { }
```

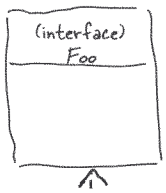
**3)**
```
public abstract class Muffi implements Wuffi { }
public class Fluffi extends Muffi { }
public interface Wuffi { }
```

**4)**
```
public class Zoop { }
public class Boop extends Zoop { }
public class Goop extends Boop { }
```

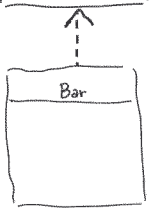
**5)**
```
public class Gamma extends Delta implements Epsilon { }
public interface Epsilon { }
public interface Beta { }
public class Alpha extends Gamma implements Beta { }
public class Delta { }
```

**Wie sieht das Bild aus?**

**1)**




**2)**




**3)**

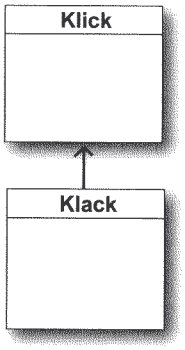
**4)**

**5)**

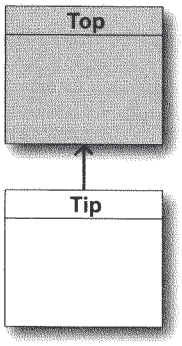
Auf der linken Seite finden Sie mehrere Sätze mit Klassendiagrammen. Ihre Aufgabe ist es, diese in gültige Java-Deklarationen umzuwandeln. Nummer 1 ist wieder schon fertig (und das war wirklich eine harte Nuss).

**Voraussetzung:**

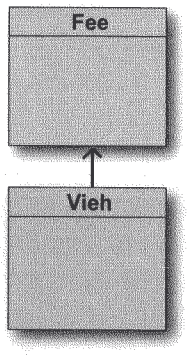
1



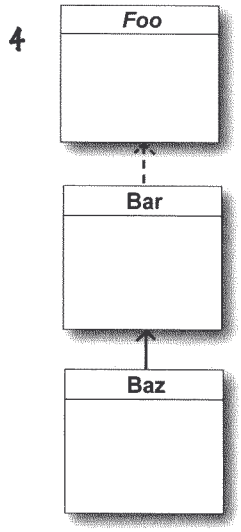

2



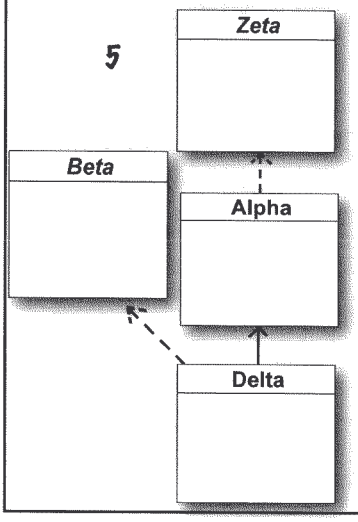

3




4




5




**Wie sieht die Deklaration aus?**

1) `public class Klick { }`

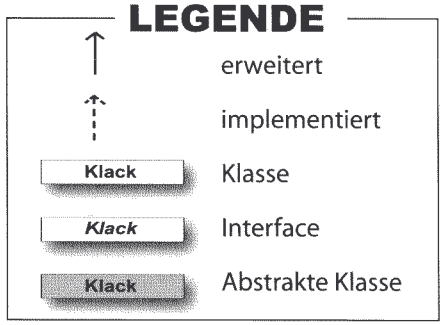
   `public class Klack extends Klick { }`

2)

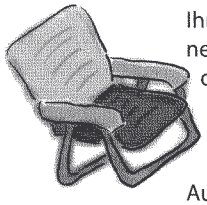
3)

4)

5)



# Pool-Puzzle



Ihre ***Aufgabe*** ist es, die Codeschnipsel aus dem Pool zu nehmen und sie in die Lücken im Code einzusetzen. Jeder Schnipsel darf auch **mehrmals** verwendet werden, und Sie werden nicht alle benötigen. Ihr ***Ziel*** ist es, einen Satz von Klassen zu erstellen, die sich kompilieren und ausführen lassen und die unten angezeigte Ausgabe hervorbringen.

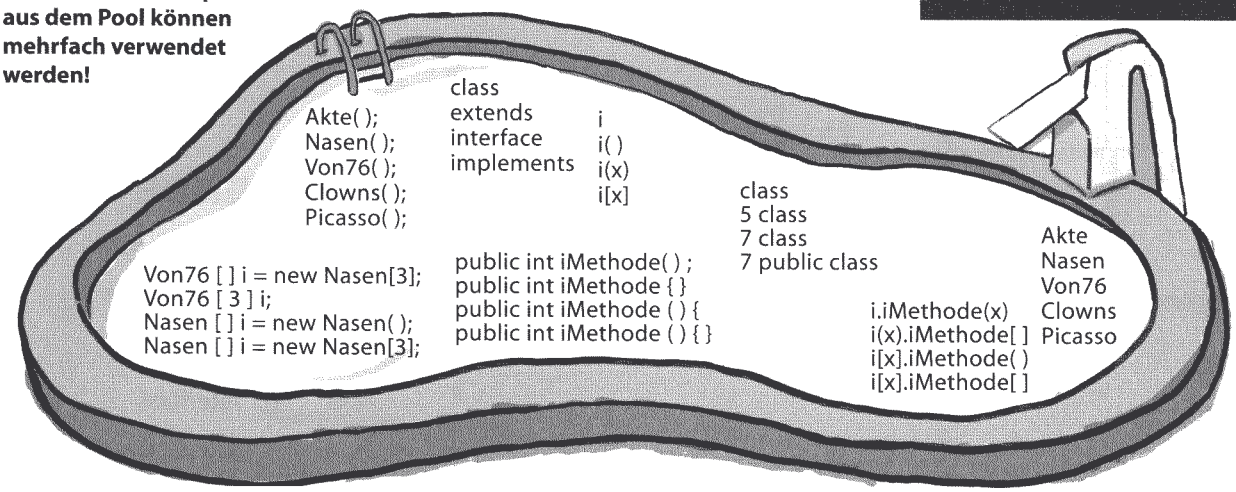
```
_____ Nasen {
  _____
}

abstract class Picasso implements _____{
  _____
    return 7;
  }
}

class _____ _____ _____ { }

class _____ _____ _____ {
  _____
    return 5;
  }
}
```

```
public _____ _____ extends Clowns {

  public static void main(String [] args) {
    _____
    i[0] = new _____
    i[1] = new _____
    i[2] = new _____
    for(int x = 0; x < 3; x++) {
      System.out.println(_____
            + " " + _____.getClass( ) );
    }
  }
}
```

**Ausgabe**

```
Datei Bearbeiten Fenster Hilfe FürchtenSieSich
%java _____
5 class Akte
7 class Clowns
_____Von76
```

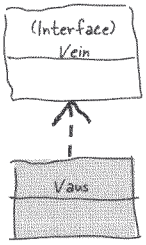
**Hinweis: Alle Schnipsel aus dem Pool können mehrfach verwendet werden!**



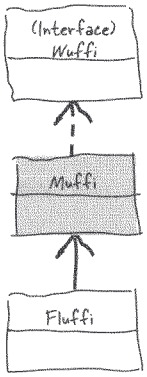
# Lösungen zu den Übungen

## Wie sieht das Bild aus?

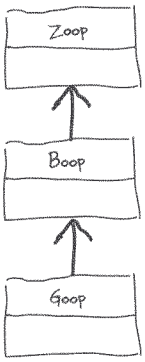
2)



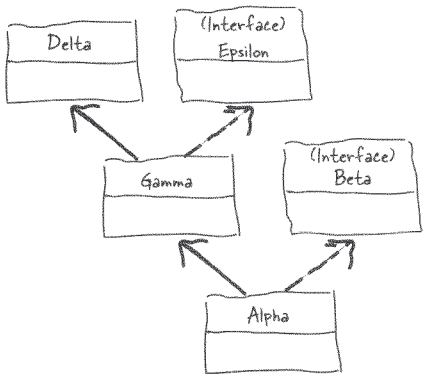

3)




4)




5)




## Wie sieht die Deklaration aus?

2)
```
public abstract class Top { }
public class Tip extends Top { }
```

3)
```
public abstract class Fee { }
public abstract class Vieh extends Fee { }
```

4)
```
public interface Foo { }
public class Bar implements Foo { }
public class Baz extends Bar { }
```

5)
```
public interface Zeta { }
public class Alpha implements Zeta { }
public interface Beta { }
public class Delta extends Alpha implements Beta { }
```

```
interface Nasen{
  public int iMethode( ) ;
}
abstract class Picasso implements Nasen {
    public int iMethode( ) {
        return 7;
    }
}
class Clowns extends Picasso { }

class Akte extends Picasso {
    public int iMethode( ) {
        return 5;
    }
}
```

```
public class Von76 extends Clowns {
   public static void main(String [] args) {
      Nasen [ ] i = new Nasen [3] ;
      i[0] = new Akte( ) ;
      i[1] = new Clowns( ) ;
      i[2] = new Von76( ) ;
      for(int x = 0; x < 3; x++) {
        System.out.println( i [x] . iMethode( )
              + " " + i [x].getClass( ) ) ;
    }
  }
}
```

**Ausgabe**

```
Datei Bearbeiten Fenster Hilfe TötetDenBarden
%java Von76
5 class Akte
7 class Clowns
7 class Von76
```

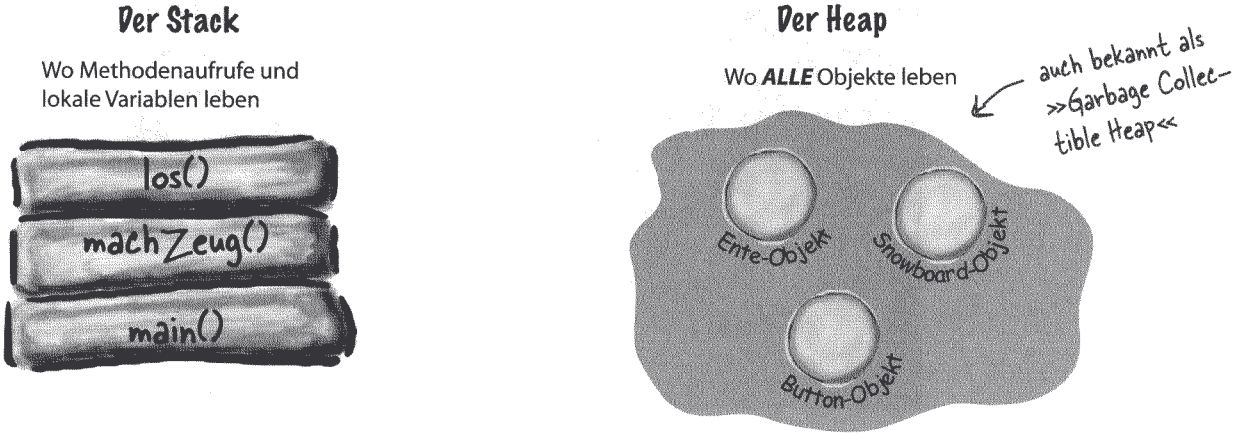
# Leben und Sterben eines Objekts

**Objekte werden geboren und Objekte sterben.** *Sie* herrschen über das Leben eines Objekts. *Sie* entscheiden, wie und wann ein Objekt **konstruiert** wird. *Sie* entscheiden, wann es **zerstört** wird. Nur *zerstören* Sie das Objekt nicht wirklich selbst, Sie *geben* es einfach auf. Aber ist es einmal aufgegeben, kann es der gnadenlose **Garbage Collector (GC)** pulverisieren und den Speicher wieder anfordern, den das Objekt belegt hat. Wenn Sie Java schreiben, erzeugen Sie Objekte. Früher oder später werden Sie einige von ihnen loslassen müssen, oder Sie riskieren, dass Ihnen das RAM ausgeht. In diesem Kapitel sehen wir uns an, wie Objekte erzeugt werden, wie sie leben, während sie leben, und wie Sie sie effektiv bewahren oder aufgeben. Das bedeutet, dass wir über den Heap, den Stack, Geltungsbereiche, Konstruktoren, Super-Konstruktoren, null-Referenzen und anderes reden werden. Warnung: Dieses Kapitel enthält Szenen zum Tod von Objekten, die für manche verstörend sein können! Besser, Sie bauen keine zu engen Bindungen auf.

# Der Stack und der Heap: wo das Leben spielt

Bevor Sie verstehen können, was wirklich passiert, wenn Sie ein Objekt erzeugen, müssen wir ein paar Schritte zurückgehen. Wir brauchen noch etwas mehr Informationen dazu, wo (und wie lange) die Dinge in Java leben. Das bedeutet, dass wir noch etwas mehr über den Stack und den Heap erfahren müssen. In Java interessieren wir (Programmierer) uns für zwei Speicherbereiche – den, in dem die Objekte leben (der Heap), und den, in dem die Methodenaufrufe und lokale Variablen leben (der Stack). Wenn eine JVM gestartet wird, wird ihr vom zu Grunde liegenden Betriebssystem ein Speicherbereich zugewiesen, den sie verwendet, um Ihr Java-Programm auszuführen. Wie *viel* Speicher das ist und ob Sie diese Menge steuern können oder nicht, ist von der JVM-Version (und der Plattform) abhängig, mit der Sie arbeiten. Aber in der Regel *müssen* Sie dazu nichts sagen. Und wenn Sie Ihr Programm ordentlich schreiben, brauchen Sie sich darüber keine Sorgen zu machen (mehr dazu gibt's etwas später).

Wir wissen, dass alle *Objekte* auf dem Garbage Collectible Heap leben, aber wir haben uns noch nicht angesehen, wo *Variablen* leben. Dies hängt davon ab, was für eine *Art* Variable sie ist. Mit »Art« meinen wir hier nicht den *Typ* (d.h. elementar oder Referenztyp). Die zwei *Arten* von Variablen, die uns jetzt interessieren, sind *Instanzvariablen* und *lokale Variablen*. Lokale Variablen werden auch als *Stack*-Variablen bezeichnet – und das ist ja schon ein freundlicher Hinweis darauf, wo sie leben.




## Instanzvariablen

**Instanzvariablen werden in einer *Klasse*, aber nicht in einer *Methode* deklariert.** Sie repräsentieren die »Felder«, die die einzelnen Objekte haben (und die bei jeder Instanz der Klasse jeweils mit unterschiedlichen Werten gefüllt sein können). Instanzvariablen leben innerhalb des Objekts, zu dem sie gehören.

```
public class Ente {

    int größe;

}
```

Jede Ente hat eine Instanzvariable >>größe<<.

## Lokale Variablen

**Lokale Variablen (einschließlich Methodenparametern) werden in einer *Methode* deklariert.** Sie sind temporär und leben nur so lange, wie die Methode auf dem Stack ist (d.h., solange die Methode noch nicht an der schließenden geschweiften Klammer angekommen ist).

```
public void foo(int x) {

    int i = x + 3;

    boolean b = true;

}
```

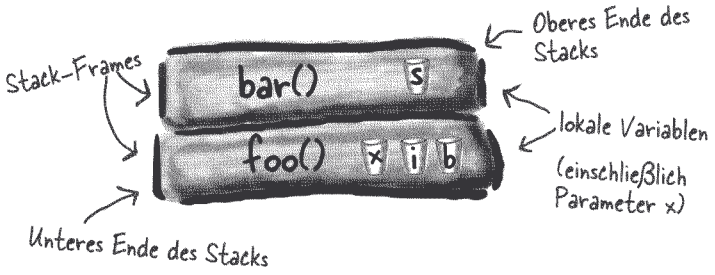
Der Parameter x und die Variablen i und b sind lokale Variablen.

# Methoden werden gestapelt

Wenn Sie eine Methode aufrufen, kommt die Methode oben auf den Aufruf-Stack. Das neue Ding, das tatsächlich auf den Stack geschoben wird, ist der Stack-*Frame*, und er enthält den Status der Methode. Dieser hält fest, welche Zeile gerade ausgeführt wird und welche Werte alle lokalen Variablen haben.

Die Methode, die sich *oben* auf dem Stack befindet, ist die auf diesem Stack gerade ablaufende Methode (im Augenblick gehen wir mal davon aus, dass es nur einen Stack gibt, aber in Kapitel 15 werden wir mehr hinzufügen). Eine Methode bleibt so lange auf dem Stack, bis die schließende geschweifte Klammer der Methode erreicht wird (die sagt, dass die Methode fertig ist). Wenn die Methode *foo()* die Methode *bar()* aufruft, wird die Methode *bar()* auf die Methode *foo()* gestapelt.

Ein Aufruf-Stack mit zwei Methoden




**Die oberste Methode auf dem Stack ist immer die, die aktuell ausgeführt wird.**

---

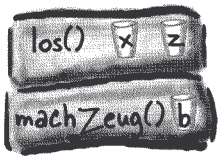
```
public void machZeug() {
   boolean b = true;
   los(4);
}
public void los(int x) {
   int z = x + 24;
   verrückt();
   // tun Sie so, als käme hier noch mehr
}
public void verrückt() {
   char c = 'a';
}
```

## Ein Stack-Szenario

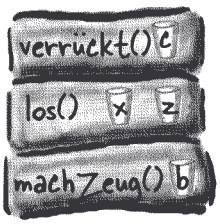
Der Code auf der linken Seite ist ein Schnipsel mit drei Methoden. (Wie der Rest der Klasse aussieht, interessiert uns nicht.) Die erste Methode (*machZeug()*) ruft die zweite Methode (*los()*) auf, und die zweite Methode ruft die dritte (*verrückt()*) auf. Im Body jeder Methode wird eine lokale Variable deklariert. Die Methode *los()* deklariert außerdem eine Parametervariable (was bedeutet, dass *los()* zwei lokale Variablen hat).

1. Code von einer anderen Klasse ruft **machZeug()** auf, und **machZeug()** kommt in einen Stack-Frame oben auf den Stack. Die boolean-Variable namens **b** kommt in den Stack-Frame für **machZeug()**.

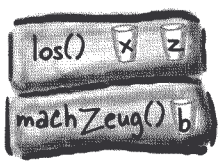
2. **machZeug()** ruft **los()** auf. **los()** wird oben auf den Stack *geschoben*. Die Variablen **x** und **z** befinden sich im Stack-Frame für **los()**.




3. **los()** ruft **verrückt()** auf. Jetzt ist **verrückt()** oben auf dem Stack, und im entsprechenden Frame befindet sich die Variable **c**.




4. **verrückt()** ist abgeschlossen, und ihr Stack-Frame wird vom Stack *geschoben*. Die Ausführung kehrt zur Methode **los()** zurück und setzt an der Zeile wieder ein, die auf den Aufruf von **verrückt()** folgt.

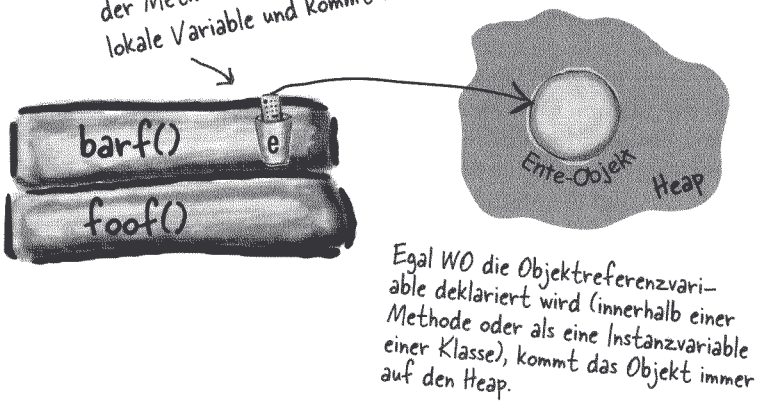


# Was ist mit lokalen Variablen, die Objekte sind?

Denken Sie daran, dass eine Variable eines nicht-elementaren Typs eine *Referenz* auf ein Objekt hält, nicht das Objekt selbst. Sie wissen bereits, wo Objekte leben – auf dem Heap. Es spielt keine Rolle, wo sie deklariert oder erzeugt werden. ***Wenn die lokale Variable eine Referenz auf ein Objekt ist, kommt nur die Variable (die Referenz/Fernsteuerung) auf den Stack.***

*Das Objekt selbst kommt trotzdem auf den Heap.*

```
public class StackRef {
   public void foof() {
      barf();
   }

   public void barf() {
      Ente e = new Ente(24);
   }
}
```




## Es gibt keine Dummen Fragen

**F: Noch mal. WARUM lernen wir diese Stack/Heap-Geschichte? Was bringt mir das? Muss ich das wirklich wissen?**

A: Dass Sie die Java-Grundlagen zu Stack und Heap kennen, ist wichtig, wenn Sie Dinge wie Geltungsbereiche von Variablen, Probleme der Objekterzeugung, Speicherverwaltung, Threads und Exception-Handling verstehen wollen. Threads und Exception-Handling werden wir in späteren Kapiteln behandeln. Aber die anderen werden Sie in diesem Kapitel kennen lernen. Sie müssen nicht wissen, *wie* Stack und Heap in einer bestimmten JVM und/oder auf einer bestimmten Plattform implementiert sind. Alles, was Sie zu Stack und Heap wissen müssen, steht auf dieser und der vorangegangenen Seite. Wenn Sie diese beiden Seiten verstanden haben, kommen Sie mit all den Themen, die von diesem Stoff abhängen, viel, viel leichter klar. Noch mal: Eines Tages werden Sie uns dafür dankbar sein, dass wir Sie mit Stacks und Heaps gefüttert haben.

## Punkt für Punkt

- Java hat zwei Speicherbereiche, die uns interessieren: den Stack und den Heap.
- Instanzvariablen werden in einer Klasse, aber außerhalb aller Methoden deklariert.
- Lokale Variablen werden in einer Methode oder als Methodenparameter deklariert.
- Alle lokalen Variablen leben in dem Stack-Frame, der zu der Methode gehört, in der die Variablen deklariert wurden.
- Objektreferenzvariablen funktionieren genau so wie Variablen eines elementaren Typs – wenn die Referenz als lokale Variable deklariert wird, kommt sie auf den Stack.
- Alle Objekte leben auf dem Heap, egal ob die Referenz eine lokale Variable oder eine Instanzvariable ist.

## Wenn lokale Variablen auf dem Stack leben, wo leben dann Instanzvariablen?

Wenn Sie new Handy() sagen, muss Java auf dem Heap Platz für das neue Handy schaffen. Aber wie *viel* Platz? Genug Platz für das Objekt! Und das heißt genug Platz, um alle Instanzvariablen des Objekts zu beherbergen. Sie haben richtig gehört: Instanzvariablen leben auf dem Heap, und zwar innerhalb der Objekte, zu denen sie gehören.

Denken Sie daran, dass die *Werte* der Instanzvariablen eines Objekts innerhalb dieses Objekts leben. Wenn alle Instanzvariablen elementare Variablen sind, erzeugt Java den Platz für die Instanzvariablen auf Basis der elementaren Typen. Ein int braucht 32 Bits, ein long 64 Bits usw. Um den Wert in elementaren Variablen kümmert sich Java nicht. Die Bit-Größe eines ints bleibt immer gleich (32 Bits), egal ob der Wert 32.000.000 oder 32 ist.

Aber was ist, wenn die Instanzvariablen *Objekte* sind? Was ist, wenn Handy eine Antenne-Instanzvariable hat?

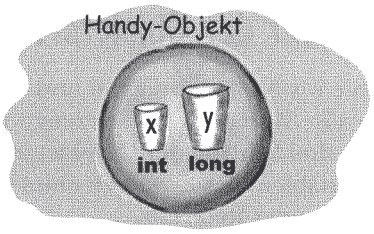
Hat das neue Objekt Instanzvariablen, die Objektreferenzen und keine elementaren Variablen sind, ist die eigentliche Frage, ob das Objekt wirklich Platz für die kompletten Objekte braucht, die es referenziert. Die Antwort ist: *nicht ganz*. Egal, worum es sich im Einzelnen handelt, Java muss Platz für die *Werte* der Instanzvariablen schaffen. Aber denken Sie daran, dass eine Referenzvariable nicht das ganze *Objekt* ist, sondern nur eine *Fernsteuerung* für das Objekt. Wenn Handy eine Instanzvariable mit dem nicht elementaren Typ Antenne hat, macht Java im Handy-Objekt nur Platz für die *Fernsteuerung* (d.h. Referenzvariable) für Antenne, aber nicht für das Antenne-*Objekt* selbst.

Aber wann erhält dann das Antenne-*Objekt* seinen Platz auf dem Heap? Dazu müssen wir erst herausfinden, *wann* das Antenne-Objekt selbst erzeugt wird. Das ist von der Deklaration der Instanzvariablen abhängig. Wenn die Instanzvariable deklariert, ihr aber kein Objekt zugewiesen wird, wird nur der Platz für die Referenzvariable (die Fernsteuerung) geschaffen.

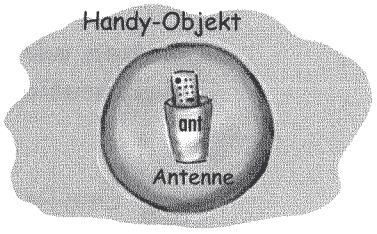
```
private Antenne ant;
```

Auf dem Heap wird kein Antenne-Objekt erzeugt, bis der Referenzvariablen ein neues Antenne-Objekt zugewiesen wird.

```
private Antenne ant = new Antenne();
```

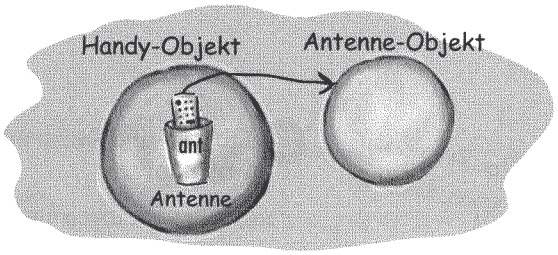



Ein Objekt mit zwei elementaren Instanzvariablen. Im Objekt ist der Platz für die beiden Variablen vorhanden.




Ein Objekt mit einer nicht elementaren Instanzvariablen – einer Referenz auf ein Antenne-Objekt, aber keinem tatsächlichen Antenne-Objekt. Das erhalten Sie, wenn Sie die Variable deklarieren, aber nicht mit einem tatsächlichen Antenne-Objekt initialisieren.

```
public class Handy {
  private Antenne ant;
}
```




Ein Objekt mit einer nicht elementaren Instanzvariablen, und der Antenne-Variablen wird ein neues Antenne-Objekt zugewiesen.

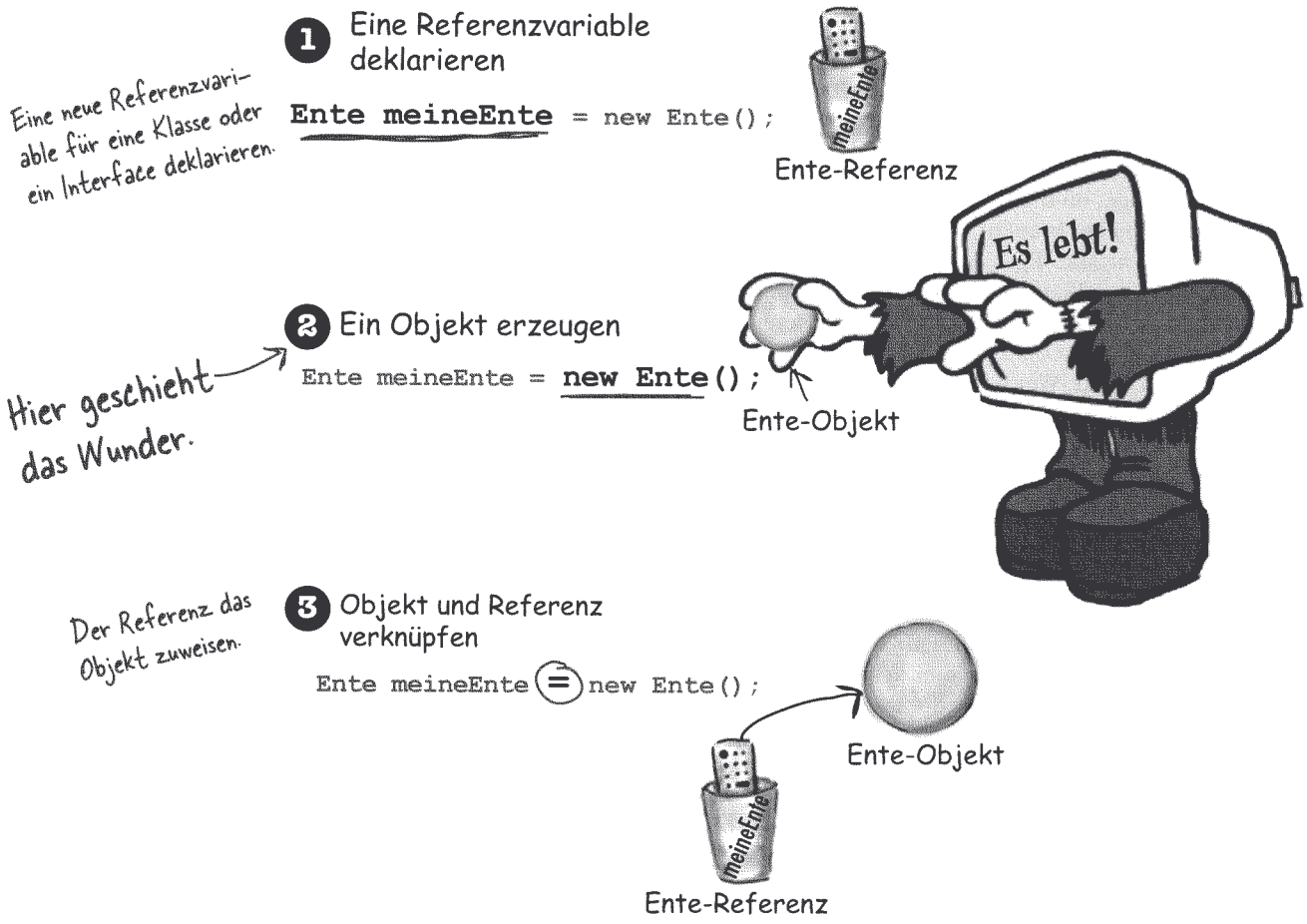
```
public class Handy {
  private Antenne ant = new Antenne();
}
```

# Das Wunder der Objekterzeugung

Jetzt, da Sie wissen, wo Variablen und Objekte leben, können wir in die mysteriöse Welt der Objekterzeugung eintauchen. Erinnern Sie sich an die drei Schritte der Objekterzeugung: die Deklaration einer Referenzvariablen, das Erzeugen eines Objekts und die Zuweisung des Objekts zu der Referenz.

Aber bis jetzt blieb Schritt zwei – in dem das Wunder geschieht und das neue Objekt »geboren« wird – ein großes Mysterium. Bereiten Sie sich darauf vor, alles zum Leben von Objekten zu erfahren. *Hoffentlich sind Sie kein Weichei.*

**Sehen wir uns die drei Schritte, Objektdeklaration, -erzeugung und -zuweisung, an:**



**Rufen wir eine Methode namens Ente() auf?**
**So *sieht* es nämlich *aus*.**

```
Ente meineEnte = new Ente();
```

Es sieht aus, als würden wir eine Methode namens Ente aufrufen, da sind schließlich auch die Klammern.

**Nein.**

**Wir rufen den Ente-*Konstruktor* auf.**

Ein Konstruktor *sieht* einer Methode sehr ähnlich, ist aber keine Methode. Er enthält den Code, der ausgeführt wird, wenn Sie **new** sagen. Anders gesagt: *den Code, der ausgeführt wird, wenn Sie ein Objekt instantiieren.*

Einen Konstruktor kann man nur mit dem Schlüsselwort **new** plus dem Namen der Klasse aufrufen. Die JVM sucht die Klasse und ruft den Konstruktor in der Klasse auf. (Okay, technisch gesehen ist das nicht die *einzige* Möglichkeit, einen Konstruktor aufzurufen. Aber es ist die einzige Möglichkeit, das *außerhalb* eines Konstruktors zu tun. Sie *können* einen Konstruktor aus einem anderen Konstruktor aufrufen. Da gibt es allerdings Einschränkungen. Aber dazu werden wir später in diesem Kapitel kommen.)

**Ein Konstruktor enthält den Code, der ausgeführt wird, wenn Sie ein Objekt instantiieren. Anders gesagt, den Code, der ausgeführt wird, wenn Sie auf dieser Klasse new sagen.**

**Jede Klasse, die Sie erstellen, hat einen Konstruktor – selbst wenn Sie selbst keinen geschrieben haben.**

**Aber wo ist dieser Konstruktor?**

**Wenn wir ihn nicht geschrieben haben, wer hat ihn dann geschrieben?**

Sie können einen Konstruktor für Ihre Klasse schreiben (und wir werden das gleich tun). Aber wenn Sie es nicht tun, ***schreibt der Compiler einen für Sie!***

So sieht der Default-Konstruktor des Compilers aus:

```
public Ente() {

}
```

**Fehlt hier irgendwas? Worin unterscheidet sich das von einer Methode?**

Der Name ist mit dem der Klasse identisch. Das ist erforderlich.

```
public Ente() {
   // hier kommt der Konstruktorcode hin
}
```

Wo ist der Rückgabetyp? Wenn das eine Methode wäre, bräuchte man zwischen >>public<< und >>Ente()<< einen Rückgabetyp.

# Eine Ente konstruieren

Das Schlüssel-Feature eines Konstruktors ist, dass er ausgeführt wird, *bevor* das Objekt einer Referenz zugewiesen werden kann. Das bedeutet, dass Sie eine Möglichkeit erhalten, einzuschreiten und Dinge zu tun, die das Objekt für den Gebrauch vorbereiten. Das Objekt hat also eine Möglichkeit, sich selbst aufzubauen, bevor jemand die Fernsteuerung für ein Objekt verwenden kann. In unserem Ente-Konstruktor machen wir nichts Nützliches. Aber er veranschaulicht die Abfolge der Ereignisse.

**Der Konstruktor gibt Ihnen eine Möglichkeit, mitten in new einzugreifen.**

```
public class Ente {

   public Ente() {
      System.out.println("Quak");
   }
}
```

Konstruktorcode.

```
public class EnteVerwenden {

   public static void main (String[] args) {
      Ente e = new Ente();
   }
}
```

Ruft den Ente-Konstruktor auf.

```
Datei Bearbeiten Fenster Hilfe Quak
% java EnteVerwenden
Quak
```

## Spitzen Sie Ihren Bleistift

Ein Konstruktor ermöglicht Ihnen, mitten in den Schritt der Objekterzeugung einzuspringen – mitten in **new**. Können Sie sich Umstände vorstellen, unter denen das nützlich sein könnte? Welche Vorbereitungen könnten im Konstruktor einer Klasse Wagen nützlich sein, wenn Wagen Teil eines RennSpiels ist? Haken Sie die ab, zu denen Ihnen ein Szenario eingefallen ist.

- ☐ Einen Zähler inkrementieren, um nachzuhalten, wie viele Objekte dieser Klasse erzeugt wurden.
- ☐ Laufzeitspezifische Daten zuweisen (Daten zu dem, was JETZT passiert).
- ☐ Den wichtigen Instanzvariablen des Objekts Werte zuweisen.
- ☐ Eine Referenz auf das Objekt, das das neue Objekt *erzeugt*, abrufen und speichern.
- ☐ Das Objekt einer ArrayList hinzufügen.
- ☐ HAS-A-Objekte erzeugen.
- ☐ _____

(Ihre eigenen Vorschläge)

## Den Zustand einer neuen Ente initialisieren

Die meisten Menschen verwenden Konstruktoren, um den Zustand eines Objekts zu initialisieren, das heißt, um Werte für die Instanzvariablen des Objekts zu erzeugen und diese zuzuweisen.

```
public Ente() {
    größe = 34;
}
```

Das ist wunderbar, wenn der *Entwickler* der Klasse Ente weiß, wie groß das Ente-Objekt sein soll. Aber was ist, wenn der Programmierer, der die Ente *verwendet*, entscheiden soll, wie groß eine bestimmte Ente ist?

Stellen Sie sich vor, die Ente hat eine Instanzvariable größe und Sie möchten, dass der Programmierer, der die Klasse Ente verwendet, die Größe der neuen Ente festlegt. Wie könnten Sie das tun?

Gut, Sie könnten der Klasse eine setGröße()-Setter-Methode geben. Aber dann hätte die Ente vorübergehend keine Größe*. Außerdem müsste der Ente-Nutzer *zwei* Anweisungen schreiben – eine, die die Ente erzeugt, und eine, die die Methode getGröße() aufruft. Der folgende Code nutzt eine Setter-Methode, um die Anfangsgröße der neuen Ente zu setzen.

```
public class Ente {
    int größe;                          ← Instanzvariable

    public Ente() {
        System.out.println("Quak");     ← Konstruktor
    }

    public void setGröße(int neueGröße) {   ← Setter-Methode
        größe = neueGröße;
    }
}
```

---

```
public class EnteVerwenden{

    public static void main (String[] args){
        Ente e = new Ente();
        e.setGröße(42);
    }
}
```

Aber hier ist etwas nicht in Ordnung. Die Ente lebt an diesem Punkt des Codes schon, hat aber noch keine Größe!* Und dann verlassen Sie sich darauf, dass der Ente-Nutzer WEISS, dass die Erzeugung einer Ente ein zweischrittiger Vorgang ist: ein Aufruf des Konstruktors und ein Aufruf des Setters.

*Instanzvariablen haben einen Default-Wert. 0 oder 0.0 bei elementaren numerischen Typen, false bei Booleschen Werten und null bei Referenzen.

### Es gibt keine Dummen Fragen

**F: Warum muss man einen Konstruktor schreiben, wenn der Compiler das für einen erledigt?**

A: Wenn Sie Code brauchen, der Sie dabei unterstützt, Ihr Objekt zu initialisieren und zur Verwendung vorzubereiten, müssen Sie Ihren eigenen Konstruktor schreiben. Vielleicht brauchen Sie beispielsweise bestimmte Benutzereingaben, bevor Sie die Vorbereitung Ihres Objekts abschließen können. Es gibt einen anderen Grund, aus dem Sie einen Konstruktor schreiben müssen, auch wenn Sie selbst keinen Konstruktorcode benötigen. Der hat mit dem Konstruktor der Superklasse zu tun, und wir werden in ein paar Minuten darüber reden.

**F: Wie kann man einen Konstruktor von einer Methode unterscheiden? Kann man auch Methoden mit dem gleichen Namen wie die Klasse haben?**

A: Java erlaubt Ihnen, eine Methode mit dem gleichen Namen wie Ihre Klasse zu erstellen. Das gibt Ihnen aber noch keinen Konstruktor. Das, was eine Methode von einem Konstruktor unterscheidet, ist der Rückgabetyp. Methoden *müssen* einen Rückgabetyp haben, aber Konstruktoren *dürfen* keinen Rückgabetyp haben.

**F: Werden Konstruktoren vererbt? Erhält man den Superklassenkonstruktor statt des Default-Konstruktors, wenn man keinen Konstruktor anbietet, aber die Superklasse einen bereitstellt?**

A: Nein. Konstruktoren werden nicht vererbt. Wir werden uns das in ein paar Seiten anschauen.

# Den Konstruktor verwenden, um die wichtigen Zustände von Ente zu initialisieren*

Sollte ein Objekt nicht verwendet werden, bis ein oder mehrere Teile seines Zustands (Instanzvariablen) initialisiert worden sind, sollten Sie niemandem ein Ente-Objekt in die Hand geben, bis Sie diese Initialisierung abgeschlossen haben! In der Regel ist es zu gefährlich, jemanden ein neues Ente-Objekt machen zu lassen – und sich eine Referenz darauf zu verschaffen –, das erst wirklich verwendungsfähig ist, wenn jemand vorbeikommt und die Methode *setGröße()* aufruft. Woher soll der Ente-Nutzer überhaupt *wissen*, dass er, nachdem er die neue Ente gemacht hat, die Setter-Methode aufrufen muss?

Der beste Ort für Initialisierungscode ist der Konstruktor. Dazu müssen Sie nur einen Konstruktor mit Argumenten erzeugen.

```
public class Ente {
   int größe;

   public Ente(int entenGröße) {
      System.out.println("Quak");

      größe = entenGröße;

      System.out.println("die Größe ist " + größe);
   }
}
```

Dem Ente-Konstruktor einen int-Parameter hinzufügen.

Den Argumentwert verwenden, um die Instanzvariable größe zu setzen.

---

```
public class EnteVerwenden {

   public static void main (String[] args) {
      Ente e = new Ente(42);
   }
}
```

Dieses Mal haben wir nur eine Anweisung. Wir machen die neue Ente und setzen ihre Größe in einer einzigen Anweisung.

Dem Konstruktor einen Wert übergeben.

```
Datei Bearbeiten Fenster Hilfe Tüt
% java EnteVerwenden
Quak
die Größe ist 42
```

*Was nicht heißen soll, dass andere Entenzustände unbedeutend wären.

# Das Erzeugen einer Ente erleichtern

## Achten Sie darauf, dass Sie einen Konstruktor haben, der kein Argument erwartet

Was passiert, wenn der Ente-Konstruktor ein Argument erwartet? Denken Sie darüber nach. Auf der vorangegangenen Seite gab es nur *einen* Ente-Konstruktor – und der erwartet ein int-Argument für die *größe* der Ente. Das muss nicht unbedingt ein großes Problem sein, erschwert es Programmierern aber, eine neues Ente-Objekt zu erzeugen, insbesondere wenn der Programmierer noch nicht *weiß*, welche Größe eine Ente haben soll. Wäre es nicht nützlich, wenn man eine Default-Größe für eine Ente hätte, damit der Benutzer auch dann eine funktionierende Ente erzeugen kann, wenn er noch keine geeignete Größe kennt?

***Stellen Sie sich vor, Sie möchten Ihren Ente-Benutzern ZWEI Möglichkeiten geben, eine Ente zu erzeugen – eine, bei der sie die Größe der Ente (als Konstruktor-Argument) angeben, und eine, bei der sie keine Größe angeben und deswegen Ihre Default-Entengröße erhalten.***

Mit einem einzigen Konstruktor können Sie das nicht sauber lösen. Denken Sie daran: Wenn eine Methode (oder ein Konstruktor – da gelten die gleichen Regeln) einen Parameter hat, *müssen* Sie ein geeignetes Argument übergeben, wenn Sie diese Methode oder diesen Konstruktor aufrufen. Sie können nicht einfach sagen: »Wenn jemand dem Konstruktor nichts übergibt, dann verwende die Default-Größe.« Das würde sich nicht einmal kompilieren lassen, wenn Sie beim Aufruf des Konstruktors kein int-Argument übergeben. Sie *könnten* irgendwas Umständliches wie das hier tun:

```
public class Ente {
   int größe;

   public Ente(int neueGröße) {
      if (neueGröße == 0) {
         größe = 27;
      } else {
         größe = neueGröße;
      }
   }
}
```

Wenn der Wert des Parameters 0 ist, der neuen Ente eine Default-Größe geben, andernfalls den Parameterwert für die Größe verwenden. KEINE sonderlich gute Lösung.

Aber das bedeutet, dass der Programmierer, der ein neues Ente-Objekt erzeugt, *wissen* muss, dass »0 übergeben« das Protokoll dafür ist, den Default-Wert zu erhalten. Ziemlich hässlich. Was ist, wenn der andere Programmierer das nicht weiß? Oder wenn er tatsächlich eine Ente mit der Größe 0 machen *möchte*? (Angenommen, eine Ente mit Größe 0 ist erlaubt. Wenn Sie keine Ente mit Größe 0 haben möchten, können Sie in den Konstruktor Validierungscode stecken, der das verhindert.) Der Punkt ist, dass es vielleicht nicht immer möglich ist, zwischen einem richtigen »ich möchte 0 als größe«-Konstruktorargument und einem »ich sende 0, damit du mir die Default-Größe lieferst, egal welche das ist«-Konstruktorargument zu unterscheiden.

**Eigentlich brauchen Sie ZWEI Möglichkeiten, eine neue Ente zu erzeugen:**

```
public class Ente2 {
   int größe;

   public Ente2() {
      // Default-Größe liefern
      größe = 27;
   }

   public Ente2(int entenGröße)
   {
      // entenGröße-Parameter
      // verwenden
      größe = entenGröße;
   }
}
```

**Machen Sie so eine Ente, wenn Sie die Größe kennen:**

```
Ente2 e = new Ente2(15);
```

**Und so machen Sie eine Ente, wenn Sie die Größe *nicht* kennen:**

```
Ente2 e2 = new Ente2();
```

Für die Zwei-Möglichkeiten-eine-Ente-zu-erzeugen-Idee brauchen Sie *zwei* Konstruktoren. Einen, der ein int annimmt, und einen, der das nicht tut. ***Wenn Sie in einer Klasse mehrere Konstruktoren haben, heißt das, dass Sie überladene Konstruktoren haben.***

## Macht der Compiler nicht immer einen Default-Konstruktor für mich? Nein!

Vielleicht erwarten Sie, dass der Compiler sieht, dass Sie keinen argumentlosen Konstruktor haben, und einen für Sie einfügt, wenn Sie *nur* einen Konstruktor mit Argumenten schreiben. Aber so funktioniert das nicht. Der Compiler kümmert sich um die Konstruktorerstellung nur dann, *wenn Sie überhaupt nichts zu Konstruktoren sagen.*

**Wenn Sie einen Konstruktor schreiben, der Argumente erwartet, und *trotzdem* noch einen Konstruktor möchten, der keine Argumente erwartet, dann müssen Sie diesen Konstruktor selbst schreiben!**

Sobald Sie einen Konstruktor anbieten, IRGENDEINEN Konstruktor, zieht sich der Compiler zurück und sagt: »Okay Kumpel, scheint, du bist jetzt für Konstruktoren verantwortlich.«

**Wenn Sie in einer Klasse mehrere Konstruktoren haben, MÜSSEN diese Konstruktoren unterschiedliche Argumentlisten haben.**

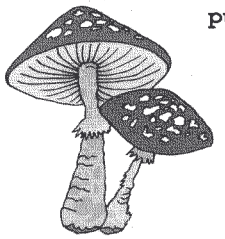
Die Argumentliste umfasst die Anzahl und die Typen der Argumente. Unterscheiden sich diese, können Sie mehrere Konstruktoren haben. Das können Sie auch mit Methoden machen, wie wir schon in Kapitel 7 gesehen haben.

## Überladene Konstruktoren bedeuten, dass Sie in Ihrer Klasse mehrere Konstruktoren haben.

### Damit sich das kompilieren lässt, müssen die verschiedenen Konstruktoren jeweils *unterschiedliche* Argumentlisten haben!

Die Klasse unten ist legal, weil alle vier Konstruktoren unterschiedliche Argumentlisten haben. Bei zwei Konstruktoren, die beide ein int annehmen, ließe sich die Klasse beispielsweise nicht kompilieren. Welche Namen Sie den Parametern geben, zählt nicht. Es sind *Variablentyp* (int, Hund usw.) und *Reihenfolge*, die entscheiden. Sie *können* zwei Konstruktoren haben, die die gleichen Typen erwarten, ***wenn die Reihenfolge anders ist.*** Ein Konstruktor, der einen String und dann ein int erwartet, ist *nicht* das Gleiche wie ein Konstruktor, der ein int und dann einen String erwartet.

Vier verschiedene Konstruktoren bieten vier verschiedene Wege, neue Pilze zu machen.



```
public class Pilz {

  public Pilz(int größe) { }

  public Pilz( ) { }

  public Pilz(boolean isGiftig) { }

  public Pilz(boolean isGiftig, int größe) { }

  public Pilz(int größe, boolean isGiftig) { }
}
```

Wenn Sie die Größe kennen, aber nicht wissen, ob der Pilz giftig ist.

Wenn Sie gar nichts wissen.

Wenn Sie wissen, ob er giftig ist oder nicht, aber die Größe nicht kennen.

Diese beiden haben die gleichen Argumente, aber in unterschiedlicher Reihenfolge, also ist das in Ordnung.

Wenn Sie wissen, ob er giftig ist oder nicht, UND auch die Größe kennen.

## Punkt für Punkt

- Instanzvariablen leben auf dem Heap in dem Objekt, zu dem sie gehören.
- Wenn die Instanzvariable eine Referenz auf ein Objekt ist, befinden sich die Referenz UND das Objekt, auf das sie verweist, auf dem Heap.
- Ein Konstruktor ist der Code, der ausgeführt wird, wenn Sie auf einer Klasse **new** sagen.
- Ein Konstruktor muss den gleichen Namen wie die Klasse und *darf* keinen Rückgabetyp haben.
- Sie können einen Konstruktor verwenden, um den Zustand (d.h. die Instanzvariablen) des Objekts zu initialisieren, das konstruiert wird.
- Wenn Sie keinen Konstruktor in Ihre Klasse stecken, fügt der Compiler einen Default-Konstruktor ein.
- Der Default-Konstruktor ist immer ein argumentloser Konstruktor.
- Wenn Sie in Ihrer Klasse einen Konstruktor – einen beliebigen Konstruktor – angeben, fügt der Compiler keinen Default-Konstruktor ein.
- Wenn Sie einen argumentlosen Konstruktor brauchen und bereits einen Konstruktor mit Argumenten geschrieben haben, müssen Sie auch den argumentlosen Konstruktor selbst schreiben.
- Geben Sie immer einen argumentlosen Konstruktor an, wenn das möglich ist. Das erleichtert es Programmierern, ein Arbeitsobjekt zu erstellen. Geben Sie Default-Werte vor.
- Überladene Konstruktoren bedeuten, dass Sie in Ihrer Klasse mehrere Konstruktoren haben.
- Überladene Konstruktoren müssen unterschiedliche Argumentlisten haben.
- Sie dürfen keine Konstruktoren mit den gleichen Argumentlisten haben. Zur Argumentliste zählen die Reihenfolge und/oder der Typ der Argumente.
- Instanzvariablen wird ein Default-Wert zugewiesen, auch wenn Sie selbst keinen expliziten Wert angeben. Die Default-Werte sind 0/0.0/false für elementare Typen und null für Referenzen.

## Spitzen Sie Ihren Bleistift

Ordnen Sie den `new Ente()`-Aufrufen die Konstruktoren zu, die ausgeführt werden, wenn die Ente instantiiert wird. Den leichten haben wir erledigt, um Ihnen auf die Sprünge zu helfen.

```
public class EnteTesten {

  public static void main(String[] args){

    int gewicht = 8;
    float dichte = 2.3F;
    String name = "Donald";
    long[] federn = {1,2,3,4,5,6};
    boolean kannFliegen = true;
    int luftGeschwindigkeit = 22;

    Ente[] e = new Ente[7];

    e[0] = new Ente();

    e[1] = new Ente(dichte, gewicht);

    e[2] = new Ente(name, federn);

    e[3] = new Ente(kannFliegen);

    e[4] = new Ente(3.3F, luftGeschwindigkeit);

    e[5] = new Ente(false);

    e[6] = new Ente(luftGeschwindigkeit, dichte);
  }
}
```

```
class Ente {

  int kilo = 6;
  float treibFähigkeit = 2.1F;
  String name = "Generisch";
  long[] federn = {1,2,3,4,5,6,7};
  boolean kannFliegen = true;
  int maxGeschwindigkeit = 25;

  public Ente() {
    System.out.println("Ententyp 1");
  }

  public Ente(boolean fliegen) {
    kannFliegen = fliegen;
    System.out.println("Ententyp 2");
  }

  public Ente(String n, long[] f) {
    name = n;
    federn = f;
    System.out.println("Ententyp 3");
  }

  public Ente(int g, float t) {
    kilo = g;
    treibFähigkeit = t;
    System.out.println("Ententyp 4");
  }

  public Ente(float dichte, int max) {
    treibFähigkeit = dichte;
    maxGeschwindigkeit = max;
    System.out.println("Ententyp 5");
  }
}
```

---

**F: Weiter oben haben Sie gesagt, dass es gut sei, wenn man argumentlose Konstruktoren bereitstellt, damit wir Default-Werte für die »fehlenden Argumente« angeben können, wenn jemand den argumentlosen Konstruktor aufruft. Aber ist es manchmal nicht unmöglich, geeignete Default-Werte vorzugeben? Kann es nicht auch sein, dass es manchmal besser ist, wenn man in einer Klasse keinen argumentlosen Konstruktor hat?**

A: Das ist richtig. Manchmal ist ein argumentloser Konstruktor sinnlos. Das sehen Sie auch in der Java-API – manche Klassen haben keinen argumentlosen Konstruktor. Die Klasse Color repräsentiert beispielsweise eine Farbe. Color-Objekte werden unter anderem dazu verwendet, die Farbe einer Bildschirmschrift oder eines GUI-Buttons zu ändern. Wenn Sie eine Color-Instanz erzeugen, steht diese Instanz für eine bestimmte Farbe (das Übliche, Tod-durch-Schokoladenbraun, Bluescreen-des-Todes-Blau, Skandalöses Rot usw.). Wenn Sie ein Color-Objekt machen, müssen Sie auf irgendeine Weise eine Farbe angeben.

```
Color f = new Color(3,45,200);
```

(Wir verwenden hier drei ints für die RGB-Werte. Wie man Color verwendet, betrachten wir später in den Swing-Kapiteln.) Denn was würden Sie erhalten, wenn Sie keine Farbe angeben? Die Programmierer der Java-API *hätten* entscheiden können, dass Sie einen wundervollen Lavendelton erhalten, wenn Sie einen argumentlosen Konstruktor aufrufen. Aber der gute Geschmack hat gesiegt. Wenn Sie versuchen, mit

```
Color f = new Color();
```

ein Color-Objekt zu erzeugen, ohne eine Farbe anzugeben, tillt der Compiler aus, weil er in der Klasse Color keinen passenden argumentlosen Konstruktor finden kann.

```
Datei Bearbeiten Fenster Hilfe Kopfeinschalten
cannot resolve symbol
:constructor Color()
location: class java.awt.Color
Color c = new Color();
          ^
1 error
```

# Kurzrückblick: Vier Dinge, die man sich zu Konstruktoren merken sollte

1. Ein Konstruktor ist der Code, der ausgeführt wird, wenn jemand auf einem Klassentyp **new** sagt.

```
Ente d = new Ente();
```

2. Ein Konstruktor muss den gleichen Namen wie die Klasse und darf ***keinen*** Rückgabetyp haben.

```
public Ente(int größe) { }
```

3. Wenn Sie Ihrer Klasse keinen Konstruktor geben, steckt der Compiler einen Default-Konstruktor in die Klasse. Der Default-Konstruktor ist immer ein argumentloser Konstruktor.

```
public Ente() { }
```

4. Sie können in einer Klasse mehrere Konstruktoren haben, wenn sich deren Argumentlisten unterscheiden. Hat man in einer Klasse mehrere Konstruktoren, heißt das, dass man überladene Konstruktoren hat.

```
public Ente() { }

public Ente(int größe) { }

public Ente(String name) { }

public Ente(String name, int größe) {
}
```

Es hat sich gezeigt, dass das ganze Gehirnjogging eine 42%ige Zunahme der Neuronenmasse bewirkt. Und Sie kennen ja den Spruch »Dicke Neuronen ...«

## KOPF-NUSS

**Was ist mit Superklassen?**

**Wenn man einen Hund erzeugt, sollte dann auch der Konstruktor von Hundeartig aufgerufen werden?**

**Und wenn die Superklasse abstrakt ist, sollte sie dann überhaupt einen Konstruktor haben?**

Das werden wir uns auf den nächsten paar Seiten ansehen. Machen Sie jetzt also Halt und denken Sie über die Implikationen von Konstruktoren und Superklassen nach.

## Es gibt keine Dummen Fragen

**F: Müssen Konstruktoren öffentlich sein?**

A: Nein. Konstruktoren können **public**, **private** oder *Default* (d.h. ohne Zugriffsmodifizierer) sein. Diesen *Default*-Zugriff sehen wir uns in Anhang B ausführlicher an.

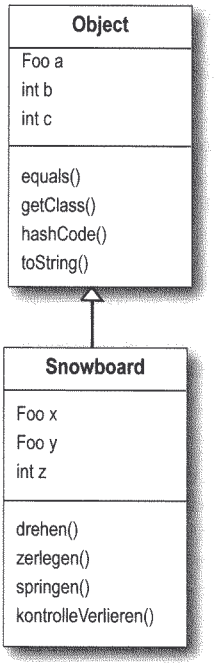
**F: Welchen Zweck kann ein privater Konstruktor überhaupt haben? Niemand könnte ihn aufrufen. Also könnte niemand neue Objekte erzeugen!**

A: Das stimmt nicht so ganz. Dass etwas als **private** markiert ist, heißt nicht, dass *niemand* darauf zugreifen kann. Das heißt nur, dass *niemand von außerhalb der Klasse* darauf zugreifen kann. Ich wette, Sie denken jetzt: »Da beißt sich der Hund doch in den Schwanz!« Nur Code aus der *gleichen* Klasse wie die Klasse-mit-dem-privaten-Konstruktor kann ein neues Objekt dieser Klasse machen. Wie aber kann man überhaupt Code dieser Klasse ausführen, wenn man nicht erst einmal ein Objekt macht? Wie kommt man jemals an irgendetwas in der Klasse? *Geduld, Grashüpfer* ... dazu kommen wir im nächsten Kapitel.

# Einen Augenblick ... wir haben im Zusammenhang mit Konstruktoren ÜBERHAUPT noch nicht über Superklassen und Vererbung gesprochen.

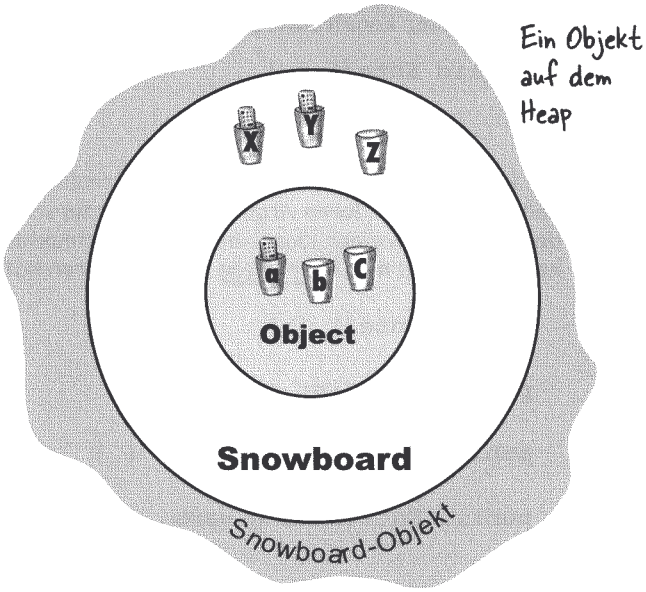
Hier beginnt das Vergnügen. Erinnern Sie sich an das letzte Kapitel? An den Teil, in dem wir uns angesehen haben, wie das Snowboard-Objekt einen inneren Kern umhüllt, der den Object-Teil der Klasse Snowboard repräsentiert? Die wichtigste Sache dort war, dass jedes Objekt nicht nur seine *eigenen* deklarierten Instanzvariablen enthält, sondern auch *alles von seinen Superklassen* (und das ist zumindest die Klasse Object, weil *jede* Klasse Object erweitert).

Wenn ein Objekt erzeugt wird (weil jemand **new** gesagt hat, es gibt ***keinen anderen Weg,*** ein Objekt zu erzeugen, als dass jemand irgendwo **new** auf dem Typ der Klasse sagt), erhält das Objekt Platz für *all* diese Instanzvariablen aus dem gesamten Vererbungsbaum. Denken Sie einen Augenblick darüber nach ... eine Superklasse könnte Setter-Methoden haben, die eine private Variable kapseln. Aber diese Variable muss *irgendwo* leben. Wenn ein Objekt erzeugt wird, ist das fast, als materialisierten sich *mehrere* Objekte – das Objekt der Klasse, auf der new gesagt wird, und ein Objekt für jede Superklasse. Theoretisch sollte man sich das allerdings besser vorstellen wie das Bild unten: Das ist so, als hätte das erzeugte Objekt *Schichten*, die die einzelnen Superklassen repräsentieren.




Object hat Instanzvariablen, die mit Zugriffsmethoden gekapselt werden. Diese Instanzvariablen werden erzeugt, wenn eine Unterklasse instantiiert wird. (Es sind keine ECHTEN Object-Variablen, aber das ist uns egal, da sie ordentlich gekapselt sind.)

Snowboard hat außerdem eigene Instanzvariablen. Wenn wir ein Snowboard-Objekt machen, brauchen wir also Platz für die Instanzvariablen beider Klassen.




Auf dem Heap gibt es nur EIN Objekt. Ein Snowboard-Objekt. Aber es enthält gleichzeitig die Snowboard-Teile seiner selbst und die Object-Teile seiner selbst. Hier müssen sich alle Instanzvariablen beider Klassen befinden.

# Die Rolle des Superklassen-Konstruktors im Leben eines Objekts.

***Alle Konstruktoren im Vererbungsbaum eines Objekts müssen ausgeführt werden, wenn Sie ein neues Objekt erzeugen.***

Lassen Sie das sacken.

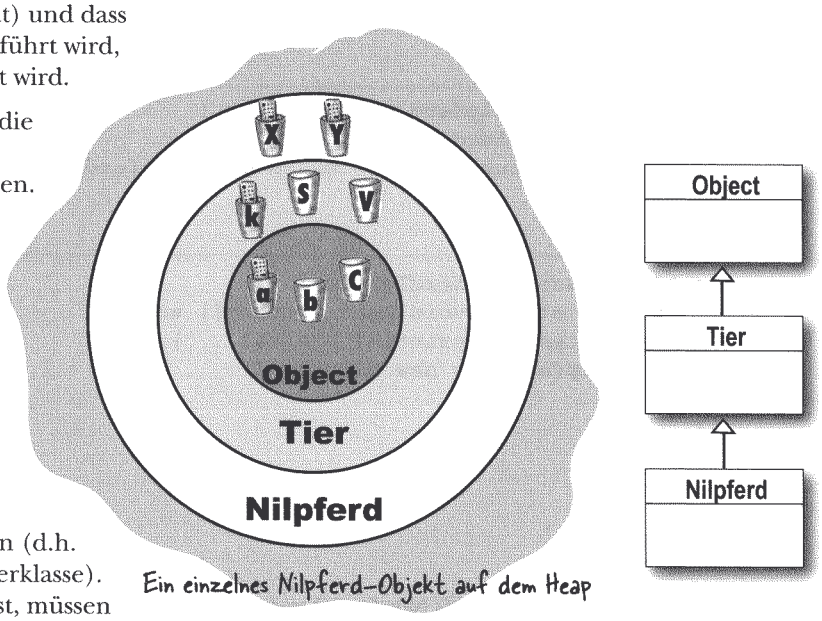
Es bedeutet, dass jede Superklasse einen Konstruktor hat (weil jede Klasse einen Konstruktor hat) und dass jeder Konstruktor in der Hierarchie ausgeführt wird, wenn ein Objekt einer Unterklasse erzeugt wird.

**new** zu sagen ist eine große Sache. Es löst die ganze Konstruktor-Kettenreaktion aus. Ja, auch abstrakte Klassen haben Konstruktoren. Obwohl man auf einer abstrakten Klasse nie new sagen kann, bleibt eine abstrakte Klasse eine Superklasse. Und deswegen wird ihr Konstruktor ausgeführt, wenn jemand eine Instanz einer konkreten Unterklasse erzeugt.

Die Super-Konstruktoren werden ausgeführt, um die Superklassenteile des Objekts aufzubauen. Denken Sie daran, dass eine Unterklasse Methoden erben kann, die vom Zustand der Superklasse abhängen (d.h. vom Wert von Instanzvariablen in der Superklasse). Damit ein Objekt vollständig ausgeformt ist, müssen auch seine Superklassenteile vollständig ausgeformt sein. Und deswegen *muss* der Superklassenkonstruktor ausgeführt werden. Alle Instanzvariablen aller Klassen im Vererbungsbaum müssen deklariert und initialisiert werden. Auch wenn die Klasse Tier Instanzvariablen hat, die Nilpferd nicht erbt (weil die Variablen beispielsweise private sind), ist das Nilpferd trotzdem noch von den Tier-Methoden abhängig, die diese Variablen *nutzen*.

Wird ein Konstruktor ausgeführt, ruft er sofort seine Superklassenkonstruktoren auf und durchläuft dabei die ganze Hierarchie bis zum Konstruktor von Object.

Auf den nächsten paar Seiten werden Sie erfahren, wie Superklassenkonstruktoren aufgerufen werden und wie Sie sie selbst aufrufen können. Sie erfahren auch, was Sie tun müssen, wenn Ihr Superklassenkonstruktor Argumente hat!




Ein einzelnes Nilpferd-Objekt auf dem Heap

**Ein neues Nilpferd-Objekt ist ein Tier und ist ein Object. Wenn Sie ein Nilpferd machen möchten, müssen Sie auch die Tier- und die Object-Teile von Nilpferd machen.**

**All das passiert in einem Prozess, der als Konstruktorverkettung bezeichnet wird.**

# Ein neues Nilpferd zu machen heißt auch, die Tier- und Object-Teile zu machen ...

```
public class Tier {
   public Tier() {
      System.out.println("Mache ein Tier");
   }
}
```

```
public class Nilpferd extends Tier {
   public Nilpferd() {
      System.out.println("Mache ein Nilpferd");
   }
}
```

```
public class NilpferdTesten {
   public static void main (String[] args) {
      System.out.println("Starte ...");
      Nilpferd n = new Nilpferd();
   }
}
```

**Spitzen Sie Ihren Bleistift**

Wie sieht die Ausgabe aus? Den Code auf der linken Seite vorausgesetzt, was wird ausgegeben, wenn Sie NilpferdTesten ausführen? A oder B?

(Die Antwort finden Sie unten.)

A

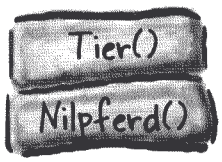
```
Datei Bearbeiten Fenster Hilfe Fluchen
% java NilpferdTesten
Starte ...
Mache ein Tier
Mache ein Nilpferd
```

B

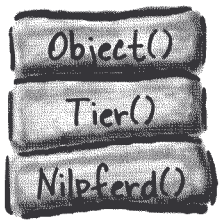
```
Datei Bearbeiten Fenster Hilfe Fluchen
% java NilpferdTesten
Starte ...
Mache ein Nilpferd
Mache ein Tier
```

1. Code von einer anderen Klasse sagt `new Nilpferd()`, und der **Nilpferd()**-Konstruktor kommt in einen Stack-Frame oben auf den Stack.

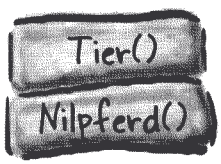
2. **Nilpferd()** ruft den Superklassenkonstruktor auf, und der **Tier()**-Konstruktor wird oben auf den Stack geschoben.




3. **Tier()** ruft den Superklassenkonstruktor auf, und der **Object()**-Konstruktor wird oben auf den Stack geschoben, da Object die Superklasse von Tier ist.




4. **Object()** wird abgeschlossen, und der entsprechende Stack-Frame wird vom Stack *gezogen*. Die Ausführung kehrt in den **Tier()**-Konstruktor zurück und fährt mit der Zeile fort, die in Tier() auf den Aufruf des Superklassenkonstruktors folgt.




Die erste Ausgabe, A. Der Nilpferd()-Konstruktor wird zuerst aufgerufen, aber der Tier-Konstruktor wird zuerst vollständig abgearbeitet.

# Wie ruft man einen Superklassenkonstruktor auf?

Sie glauben vielleicht, dass man beispielsweise irgendwo in einem Ente-Konstruktor Tier() aufruft, wenn Ente Tier erweitert. Aber so funktioniert das nicht:

```
public class Ente extends Tier {
   int größe;

   public Ente(int neueGröße) {
      Tier();
      größe = neueGröße;
   }
}
```

SCHLECHT! → `Tier();` ← NEIN! Das ist nicht zulässig!

Einen Superkonstruktor kann man nur mit ***super()*** aufrufen. Das ist richtig – ***super()*** ruft den ***Superkonstruktor*** auf.

Was für ein seltsamer Zufall!

```
public class Ente extends Tier {
   int größe;

   public Ente(int neueGröße) {
      super();
      größe = neueGröße;
   }
}
```

`super();` ← Sagen Sie einfach super()!

Wenn Sie in Ihrem Konstruktor *super()* aufrufen, kommt der Superklassenkonstruktor oben auf den Stack. Und was meinen Sie, macht der Superklassenkonstruktor? *Er ruft seinen Superklassenkonstruktor auf.* Und so geht das weiter, bis oben auf dem Stack der Object-Konstruktor liegt. Wenn *Object()* fertig ist, wird er vom Stack genommen, und es befindet sich das oben, was auf dem Stack als Nächstes kommt (der Unterklassenkonstruktor, der *Object()* aufgerufen hat). *Dieser* Konstruktor wird beendet und so geht das weiter, bis der ursprüngliche Konstruktor wieder oben auf dem Stack liegt, wo *er* jetzt abgearbeitet werden kann.

## Und wie kommt es, dass wir davongekommen sind, ohne das zu tun?

Wahrscheinlich haben Sie sich das schon gedacht.

### Unser guter Freund, der Compiler, schiebt den Aufruf von *super()* ein, wenn Sie das nicht tun.

Der Compiler ist am Konstruktormachen also auf *zweierlei* Weise beteiligt:

### ① Wenn Sie *keinen* Konstruktor angeben:

Der Compiler fügt einen ein, der so aussieht:

```
public KlassenName() {
      super();
}
```

### ② Wenn Sie einen Konstruktor *angeben*, aber den Aufruf von super() *nicht* einfügen:

Der Compiler fügt den Aufruf von super() in jeden Ihrer überladenen Konstruktoren ein.*

Der vom Compiler eingefügte Aufruf sieht so aus:

```
super();
```

Er sieht immer so aus. Der vom Compiler eingefügte Aufruf von *super()* ist immer ein argumentloser Aufruf. Wenn die Superklasse überladene Konstruktoren hat, wird nur der argumentlose Konstruktor aufgerufen.

*Es sei denn, der Konstruktor ruft einen anderen überladenen Konstruktor auf (das werden Sie in ein paar Seiten sehen).

# Kann das Kind vor den Eltern leben?

Wenn Sie sich eine Superklasse als Eltern des Unterklassenkindes vorstellen, können Sie herausfinden, wer zuerst bestehen muss. ***Die Superklassenteile eines Objekts müssen vollständig ausgeformt (vollständig aufgebaut) sein, bevor die Unterklassenteile konstruiert werden können.*** Denken Sie daran, dass das Unterklassenobjekt von Dingen abhängig sein könnte, die es von seiner Superklasse erbt. Deswegen ist es wichtig, dass diese geerbten Dinge zuerst fertig gestellt werden. Daran führt kein Weg vorbei. Der Superklassenkonstruktor muss vor dem Unterklassenkonstruktor fertig sein.

Schauen Sie sich die Stack-Abfolge auf Seite 252 noch einmal an. Dann sehen Sie, dass der Nilpferd-Konstruktor zwar *zuerst* aufgerufen wird (er ist das erste Ding auf dem Stack), aber *zuletzt* fertig ist! Jeder Unterklassenkonstruktor ruft sofort den Konstruktor seiner eigenen Superklasse auf, bis sich der Konstruktor von Object oben auf dem Stack befindet. Ist der Konstruktor von Object fertig, gehen wir den Stack wieder nach unten zum Konstruktor von Tier. Erst nachdem der Konstruktor von Tier abgeschlossen ist, kommen wir endlich zurück, um den Rest des Nilpferd-Konstruktors zu erledigen. Deswegen:

**Der Aufruf von super() muss die *erste* Anweisung in jedem Konstruktor sein!***

Mögliche Konstruktoren für die Klasse Blub

✓
```
public Blub() {
    super();
}
```
Die beiden sind okay, weil der Programmierer den Aufruf von super() explizit als erste Anweisung angegeben hat.

✓
```
public Blub(int i) {
    super();
    größe = i;
}
```

✓
```
public Blub() {
}
```
Diese beiden sind okay, weil der Compiler den Aufruf von super() als erste Anweisung einfügt.

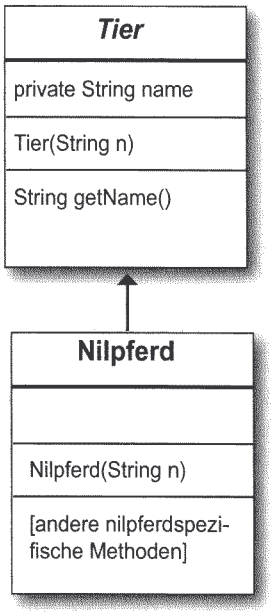
✓
```
public Blub(int i) {
    größe = i;
}
```

✗
```
public Blub(int i) {
    größe = i;
    super();
}
```
NEIN!! Das lässt sich nicht kompilieren! Sie können den Aufruf von super() nicht explizit nach etwas anderem angeben.

*Es gibt eine Ausnahme zu dieser Regel. Sie werden sie auf Seite 256 kennen lernen.

# Superklassenkonstruktoren mit Argumenten

Was ist, wenn der Superklassenkonstruktor Argumente hat? Kann man mit dem Aufruf von *super()* etwas übergeben? Natürlich. Wenn man das nicht könnte, könnte man Klassen, die keinen argumentlosen Konstruktor haben, überhaupt nicht erweitern. Stellen Sie sich folgendes Szenario vor: Alle Tiere haben einen Namen. In der Klasse Tier gibt es eine *getName()*-Methode, die den Wert der Instanzvariablen *name* zurückliefert. Die Instanzvariable ist als private markiert, aber die Unterklasse (in diesem Fall Nilpferd) erbt die Methode *getName()*. Das Problem ist also folgendes: Nilpferd hat (durch Vererbung) eine *getName()*-Methode, aber keine Instanzvariable *name*. Nilpferd ist davon abhängig, dass sein Tier-Teil die Instanzvariable name festhält und ihren Wert zurückliefert, wenn jemand *getName()* auf einem Nilpferd-Objekt aufruft. Aber ... wie erhält der Tier-Teil den Namen? Die einzige Referenz, die Nilpferd auf seinen Tier-Teil hat, ist über *super()*. Das muss also der Ort sein, über den Nilpferd den Namen des Nilpferds an seinen Tier-Teil schickt, damit der Tier-Teil ihn in der privaten Instanzvariablen *name* speichern kann.




```
public abstract class Tier {
   private String name;
   public String getName() {
      return name;
   }
   public Tier(String derName) {
      name = derName;
   }
}
```

Alle Tiere (einschließlich der Unterklassen) haben einen Namen.

Eine Getter-Methode, die Nilpferd erbt.

Der Konstruktor, der den Namen entgegennimmt und der Instanzvariablen name zuweist.

```
public class Nilpferd extends Tier {

   public Nilpferd(String name) {
      super(name);
   }
}
```

Der Nilpferd-Konstruktor nimmt einen Namen.

und sendet ihn den Stack hinauf zum Tier-Konstruktor.

```
public class NilpferdMachen{
   public static void main(String[] args) {
      Nilpferd h = new Nilpferd("Kurt");
      System.out.println(h.getName());
   }
}
```

Ein Nilpferd machen und dabei den Namen »Kurt« an den Nilpferd-Konstruktor übergeben.

Dann die geerbte getName()-Methode von Nilpferd aufrufen.

```
Datei Bearbeiten Fenster Hilfe Verstecken
%java NilpferdMachen
Kurt
```

# Einen überladenen Konstruktor aus einem anderen Konstruktor aufrufen

Was ist, wenn man überladene Konstruktoren hat, die zwar mit unterschiedlichen Argumenttypen umgehen können, sonst aber genau das Gleiche machen? Sie wissen, dass Sie *doppelten Code* vermeiden wollen, der in jedem der Konstruktoren sitzt (grausam bei der Wartung usw.). Also würden Sie gern den Großteil des Konstruktorcodes (einschließlich des super()-Aufrufs) in einen *einzigen* der überladenen Konstruktoren stecken. Und jeder Konstruktor, egal welcher zuerst aufgerufen wird, soll dann **Den Richtigen Konstruktor** aufrufen und ihn die Aufbauarbeit erledigen lassen. Das ist leicht: Sagen Sie einfach *this()*. Oder *this(einString)*. Oder *this(27, x)*. Mit anderen Worten: Stellen Sie sich einfach vor, dass das Schlüsselwort *this* eine Referenz auf **das aktuelle Objekt** ist.

*this()* können Sie nur in einem Konstruktor sagen, und es muss die erste Anweisung im Konstruktor sein!

Aber das ist ein Problem, oder? Eben haben wir gesagt, dass super() die erste Anweisung im Konstruktor sein muss. Das bedeutet, dass Sie die Wahl haben.

**Jeder Konstruktor kann einen Aufruf von super() oder von this() enthalten, aber niemals beides gleichzeitig!**

**Nutzen Sie this( ), um einen Konstruktor aus einem anderen überladenen Konstruktor in der gleichen Klasse aufzurufen.**

**Der Aufruf von this( ) kann nur in einem Konstruktor verwendet werden und muss die erste Anweisung in einem Konstruktor sein.**

**Ein Konstruktor kann entweder einen Aufruf von super( ) ODER einen Aufruf von this( ) haben, aber nie beides!**

```
class Mini extends Auto {

    Color farbe;

    public Mini() {
        this(Color.Red);
    }

    public Mini(Color f) {
        super("Mini");
        farbe = f;
        // mehr Initialisierungscode
    }

    public Mini(int größe) {
        this(Color.Red);
        super(größe);
    }
}
```

Der argumentlose Konstruktor gibt eine Default-Farbe an und ruft den überladenen richtigen Konstruktor auf (den, der super() aufruft).

Das ist Der Richtige Konstruktor, der Die Richtige Arbeit der Initialisierung des Objekts leistet (einschließlich des Aufrufs von super()).

Funktioniert nicht!! super() und this() können nicht im gleichen Konstruktor stehen, weil beide jeweils die erste Anweisung im Konstruktor sein müssen!

```
Datei Bearbeiten Fenster Hilfe Fahren
javac Mini.java

Mini.java:16: call to super must
be first statement in constructor
        super();
        ^
```

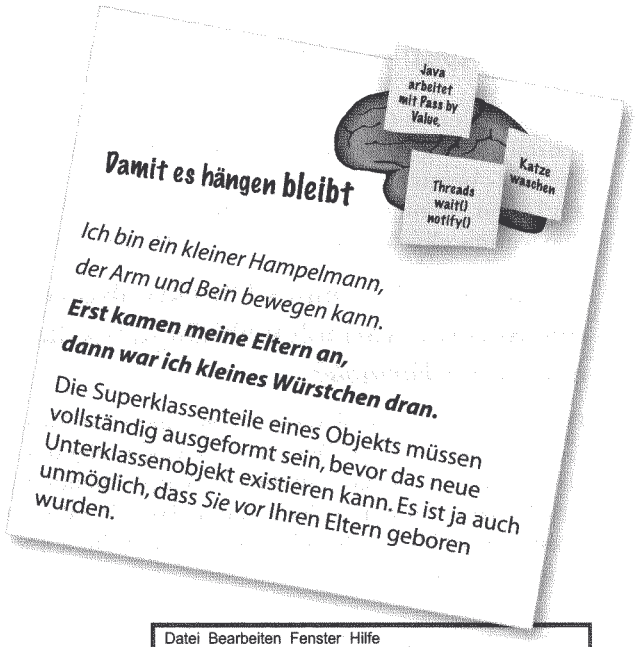
## Spitzen Sie Ihren Bleistift

Einige Konstruktoren in der Klasse SohnVonBuh lassen sich nicht kompilieren. Können Sie herausfinden, welche der Konstruktoren nicht zulässig sind? Ordnen Sie die Compiler-Fehler den SohnVonBuh-Konstruktoren zu, die sie verursacht haben, indem Sie eine Linie vom Compiler-Fehler zu dem entsprechenden »schlechten« Konstruktor ziehen.

```
public class Buh {
   public Buh(int i) { }
   public Buh(String s) { }
   public Buh(String s, int i) { }
}
```

---

```
class SohnVonBuh extends Buh {

   public SohnVonBuh() {
      super("Buh");
   }

   public SohnVonBuh(int i) {
      super("Fred");
   }

   public SohnVonBuh(String s) {
      super(42);
   }

   public SohnVonBuh(int i, String s) {
   }

   public SohnVonBuh(String a, String b, String c) {
      super(a,b);
   }

   public SohnVonBuh(int i, int j) {
      super("Mann", j);
   }

   public SohnVonBuh(int i, int x, int y) {
      super(i, "Stern");
   }
}
```




```
Datei Bearbeiten Fenster Hilfe
%javac SohnVonBuh.java

cannot resolve symbol

symbol : constructor Buh
(java.lang.String,java.la
ng.String)
```

```
Datei Bearbeiten Fenster Hilfe BlaBlaBla
%javac SohnVonBuh.java

cannot resolve symbol

symbol : constructor Buh
(int,java.lang.String)
```

```
Datei Bearbeiten Fenster Hilfe IchHöreNichtZu
%javac SohnVonBuh.java

cannot resolve symbol

symbol:constructor Buh()
```

# Wir wissen jetzt, wie ein Objekt geboren wird. Aber wie lange *lebt* ein Objekt?

Das Leben eines *Objekts* ist vollkommen vom Leben der Referenzen abhängig, die es referenzieren. Wenn die Referenz als »lebend« betrachtet wird, ist das Objekt auf dem Heap noch am Leben. Wenn die Referenz stirbt (und was das heißt, werden wir uns gleich ansehen), stirbt auch das Objekt.

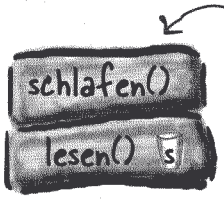
## Wenn das Leben eines Objekts also vom Leben der Referenzvariablen abhängt, wie lange lebt dann eine *Variable*?

Das hängt davon ab, ob die Variable eine *lokale* Variable oder eine *Instanzvariable* ist. Der folgende Code zeigt das Leben einer lokalen Variablen. In diesem Beispiel handelt es sich um eine Variable für einen elementaren Typ, aber die Lebensdauer einer Variablen ist bei elementaren Variablen und Referenzvariablen gleich.

```
public class LebenstestEins {

   public void lesen() {
      int s = 42;
      schlafen();
   }

   public void schlafen() {
      s = 7;
   }
}
```

»s« ist nur in der Methode lesen() gültig und kann deswegen nirgendwo sonst verwendet werden.

SCHLECHT!! Hier ist die Verwendung von »s« nicht zulässig!




schlafen() kann die Variable »s« nicht sehen. Da sie sich nicht im Stack-Frame von schlafen() befindet, weiß schlafen() nichts über diese Variable.

Die Variable »s« lebt, aber sie ist nur in der Methode lesen() gültig. Wenn schlafen() fertig ist und lesen() sich oben auf dem Stack befindet und wieder ausgeführt wird, kann lesen() »s« immer noch sehen. Wenn lesen() fertig ist und vom Stack genommen wird, ist »s« tot und sieht sich die digitalen Radieschen von unten an.

### ① Eine lokale Variable lebt nur in der Methode, die die Variable deklariert hat.

```
public void lesen() {
   int s = 42;
   // 's' kann nur innerhalb
   // dieser Methode verwendet
   // werden. Wenn diese Methode
   // endet, verschwindet 's'
   // vollständig.
}
```

Die Variable »**s**« kann *nur* in der Methode *lesen()* verwendet werden. Anders gesagt: ***Der Geltungsbereich dieser Variablen ist auf diese Methode beschränkt.*** Kein anderer Code in der Klasse (oder in einer anderen Klasse) kann »**s**« sehen.

### ② Eine Instanzvariable lebt so lange wie das Objekt. Solange das Objekt lebt, leben seine Instanzvariablen.

```
public class Leben {
   int größe;

   public void setGröße(int s) {
      größe = s;
      // 's' verschwindet am
      // Ende dieser Methode,
      // aber 'größe' kann
      // überall in dieser
      // Klasse verwendet
      // werden.
   }
}
```

Die Variable »**s**« (diesmal ein Methodenparameter) ist nur innerhalb der Methode setGröße() in Geltung. Aber die Instanzvariable »größe« ist, unabhängig vom Leben der *Methode*, so lange in Geltung, wie das *Objekt* lebt.

## Der Unterschied zwischen **Leben** und **Geltung** bei lokalen Variablen:

### Leben

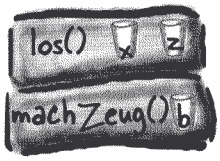
Eine lokale Variable *lebt*, solange ihr Stack-Frame auf dem Stack ist. Mit anderen Worten: *bis die Methode abgeschlossen ist.*

### Geltung

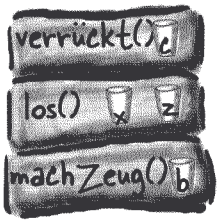
Eine lokale Variable ist nur innerhalb der Methode in *Geltung*, in der die Variable deklariert wurde. Wenn diese Methode eine andere Methode aufruft, ist die Variable zwar immer noch am Leben, aber nicht mehr in Geltung, bis die Methode wieder aufgenommen wird. ***Sie können eine Variable nur innerhalb ihres Geltungsbereichs verwenden.***

```
public void machZeug() {
   boolean b = true;
   los(4);
}
public void los(int x) {
   int z = x + 24;
   verrückt();
   // stellen Sie sich hier mehr Code vor
}
public void verrückt() {
   char c = 'a';
}
```

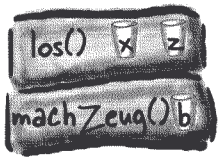
1 ***machZeug()*** **kommt auf den Stack. Die Variable »b« lebt und ist gültig.**




2 ***los()*** **kommt oben auf den Stack. »x« und »z« leben und sind gültig, und »b« lebt, ist aber *nicht* gültig.**




3 ***verrückt()*** **wird auf den Stack geschoben, und jetzt lebt »c« und ist gültig. Die anderen drei Variablen leben, sind aber nicht in Geltung.**




4 ***verrückt()*** **wird abgeschlossen und vom Stack genommen, »c« ist also nicht mehr gültig *und tot*. Wenn los() wieder aufgenommen wird, wo es verlassen wurde, sind »x« und »z« wieder lebendig und in Geltung. Die Variable »b« lebt immer noch, ist aber nicht in Geltung (bis los() fertig ist).**

Solange eine lokale Variable lebt, bleibt ihr Zustand bestehen. Solange die Methode machZeug() auf dem Stack ist, behält die Variable »b« ihren Wert. Aber die Variable »b« kann nur verwendet werden, solange sich der Stack-Frame von machZeug() oben auf dem Stack befindet. Das heißt, dass Sie eine lokale Variable *nur* verwenden können, solange die Methode dieser lokalen Variablen tatsächlich ausgeführt wird (und nicht einfach nur darauf wartet, dass über ihr liegende Stack-Frames fertig sind).

# Was ist mit Referenzvariablen?

Für elementare Variablen und Referenzvariablen gelten die gleichen Regeln. Eine Referenzvariable kann nur innerhalb ihres Geltungsbereichs verwendet werden. Das heißt, Sie können die Fernsteuerung für ein Objekt nur verwenden, wenn Sie eine Referenzvariable haben, die in Geltung ist. Die *eigentliche* Frage ist:

## »Wie beeinflusst das *Variablenleben* das *Objektleben*?«

Ein Objekt lebt, solange es lebende Referenzen darauf gibt. Wenn eine Referenzvariable nicht mehr in Geltung, aber immer noch am Leben ist, bleibt das Objekt auf dem Heap, das sie *referenziert*, am Leben. Und dann müssen Sie fragen: »Was passiert, wenn der Stack-Frame, der die Referenz hält, am Ende der Methode vom Stack genommen wird?«

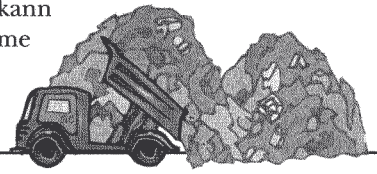
War das die *einzige* Referenz auf das Objekt, wurde dieses Objekt auf dem Heap aufgegeben. Die Referenzvariable hat sich mit dem Stack-Frame aufgelöst, und das aufgegebene Objekt ist jetzt *offiziell* GC-Futter. Entscheidend ist, dass man weiß, wann ein Objekt zur ***Garbage Collection ausgewählt werden darf.***

Wenn ein Objekt zur Garbage Collection (GC) ausgewählt werden darf, müssen Sie sich nicht mehr darum kümmern, den Speicher zurückzufordern, den das Objekt benötigt hat. Steht Ihrem Programm nur noch wenig Speicherplatz zur Verfügung, zerstört die GC einige oder alle zur Garbage Collection auswählbaren Objekte und verhindert so, dass Ihnen das RAM ausgeht. Ihnen kann immer noch der Speicher ausgehen, aber *nicht* bevor alle zur Garbage Collection auswählbaren Objekte zur Kippe gefahren wurden. Ihr Job ist es sicherzustellen, dass Sie Objekte aufgeben (und so zur Garbage Collection auswählbar machen), wenn Sie sie nicht mehr benötigen, damit die GC etwas hat, das sie zurückfordern kann. Wenn Sie an Ihren Objekten kleben, kann die GC Ihnen nicht helfen. Dann riskieren Sie, dass Ihre Programme einen grausamen Tod sterben.

**Das Leben eines Objekts hat keinen Wert, keine Bedeutung, keinen Sinn, wenn keiner mehr eine Referenz darauf hält.**

**Wenn Sie nicht zum Objekt gelangen können, können Sie es nicht bitten, etwas für Sie zu tun. Dann ist es nichts anderes als fetter Haufen Bitmüll.**

**Aber wenn ein Objekt unerreichbar ist, findet der Garbage Collector das heraus. Früher oder später wird dieses Objekt entsorgt.**



**Ein Objekt ist zur GC auswählbar, wenn die letzte lebende Referenz darauf weg ist.**

**Die drei Möglichkeiten, Referenzen auf ein Objekt loszuwerden:**

1. Eine Referenz verliert dauerhaft ihre Geltung.

```
void go() {
  Leben z = new Leben();
}
```

Die Referenz z stirbt am Ende der Methode.

2. Der Referenz wird ein anderes Objekt zugewiesen.

```
Leben z = new Leben();
z = new Leben();
```

Das erste Objekt wird aufgegeben, wenn z auf ein neues Objekt »umprogrammiert« wird.

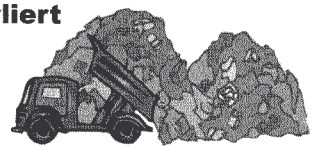
3. Die Referenz wird explizit auf null gesetzt.

```
Leben z = new Leben();
z = null;
```

Das erste Objekt wird aufgegeben, wenn die Programmierung von z »aufgehoben wird«.

## Objekt-Killer Nr. 1

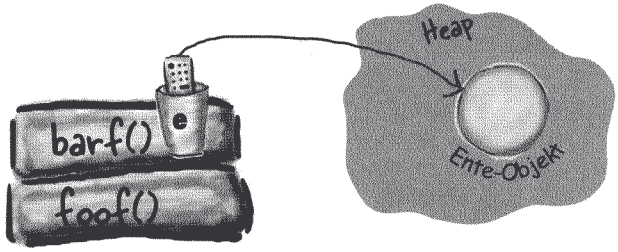
**Die Referenz verliert dauerhaft ihre Geltung.**



```
public class StackRef {
   public void foof() {
      barf();
   }
   public void barf() {
      Ente e = new Ente();
   }
}
```

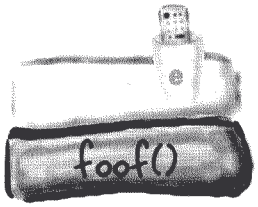
1. *foof()* wird auf den Stack geschoben. Keine Variablen sind deklariert.

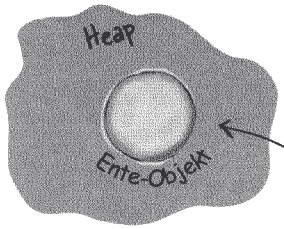
2. *barf()* wird auf den Stack geschoben. Dort deklariert es eine Referenzvariable und erzeugt ein neues Objekt, das dieser Referenz zugewiesen wird. Das Objekt wird auf dem Heap erzeugt, und die Referenz lebt und ist in Geltung.




Die neue Ente kommt auf den Heap, und solange barf() läuft, lebt die Referenz >>e<< und ist gültig. Die Ente wird also als lebend betrachtet.

3. *barf()* ist beendet und wird vom Stack genommen. Der Stack-Frame wird aufgelöst, und »e« ist jetzt tot und weg. Die Ausführung kehrt zu *foof()* zurück, aber *foof()* kann »e« nicht verwenden.

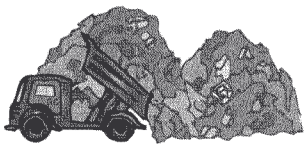


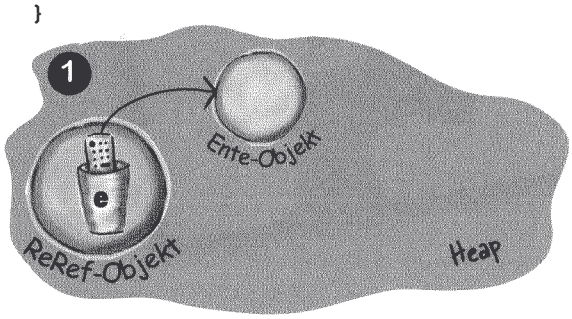

Oje! Als der Stack-Frame für barf() vom Stack geblasen wurde, ging die Variable >>e<<, und damit wurde die Ente aufgegeben. Garbage Collector-Futter.

## Objekt-Killer Nr. 2

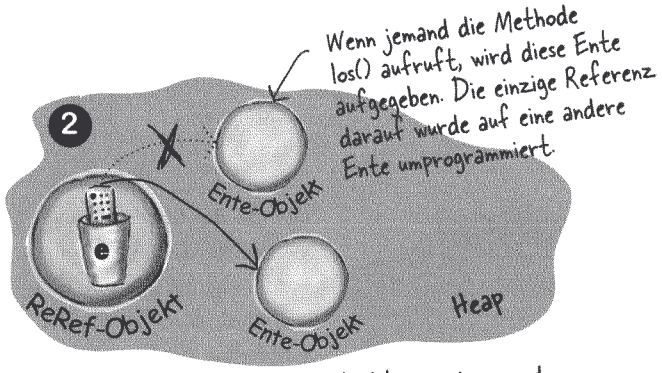
**Der Referenz ein anderes Objekt zuweisen.**



```
public class ReRef {

    Ente e = new Ente();

    public void los() {
      e = new Ente();
    }
}
```

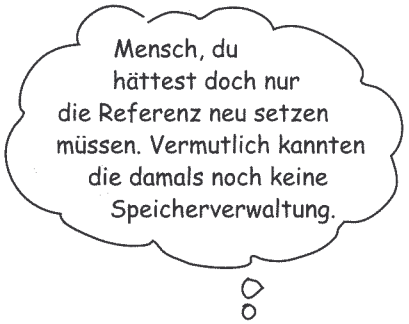


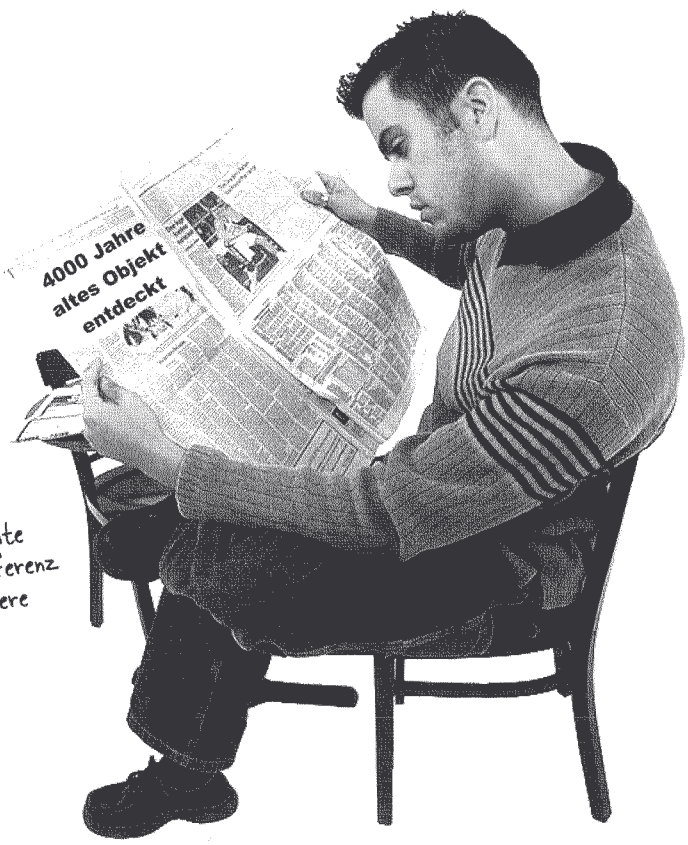

Die neue Ente kommt auf den Heap und wird von >>e<< referenziert. Da >>e<< eine Instanzvariable ist, lebt die Ente, solange das ReRef-Objekt lebt, das sie instantiiert hatte. Es sei denn ...




Wenn jemand die Methode los() aufruft, wird diese Ente aufgegeben. Die einzige Referenz darauf wurde auf eine andere Ente umprogrammiert.

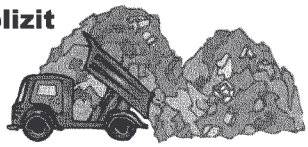
>>e<< erhält ein neues Ente-Objekt zugewiesen und gibt damit das ursprüngliche (erste) Ente-Objekt auf. Die erste Ente ist jetzt so gut wie tot.



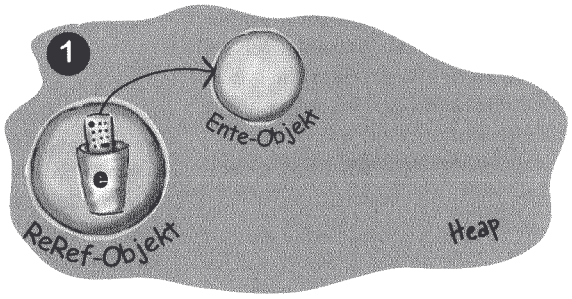



## Objekt-Killer Nr. 3

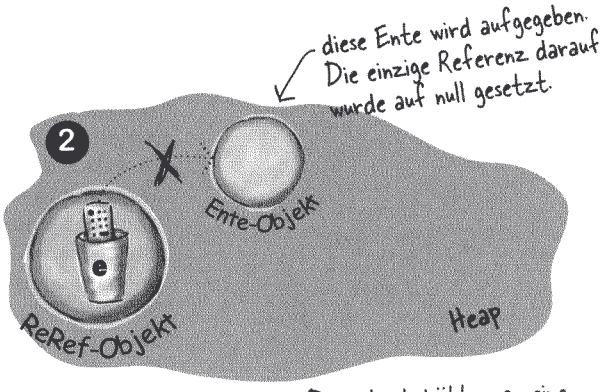
### Die Referenz explizit auf null setzen



```
public class ReRef {

    Ente e = new Ente();

    public void los() {
      e = null;
    }
}
```




Die neue Ente kommt auf den Heap und wird von >>e<< referenziert. Da >>e<< eine Instanzvariable ist, lebt die Ente, solange das ReRef-Objekt lebt, das sie instantiiert hatte. Es sei denn ...




>>e<< wird auf null gesetzt. Das ist, als hätte man eine Fernsteuerung, die auf nichts programmiert ist. Sie dürfen auf >>e<< noch nicht einmal den Punktoperator verwenden, bis sie neu programmiert wird (ihr ein Objekt zugewiesen wird).

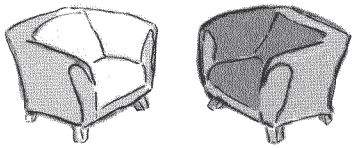
### Die Bedeutung von `null`

Wenn Sie eine Referenz auf `null` setzen, heben Sie die Programmierung der Fernsteuerung auf. Mit anderen Worten: Sie haben eine Fernsteuerung, aber am anderen Ende gibt es keinen Fernseher. Eine null-Referenz hat Bits, die »null« repräsentieren (*wir* wissen nicht, was diese Bits sind, und kümmern uns auch nicht darum, das überlassen wir der JVM).

Wenn Sie im richtigen Leben eine nicht programmierte Fernsteuerung haben, passiert nichts, wenn Sie auf eine der Tasten drücken. Aber in Java können Sie auf einer null-Referenz nicht auf die Tasten drücken (d.h. den Punktoperator verwenden). Sie hätten vielleicht gern, dass etwas für Sie bellt, aber die JVM weiß (das ist ein Laufzeitproblem, kein Compiler-Fehler), dass leider kein Hund da ist, der das machen könnte!

**Wenn Sie auf einer null-Referenz den Punktoperator verwenden, erhalten Sie zur Laufzeit eine NullPointerException.** Im Kapitel *Riskantes Verhalten* werden Sie alles zu Exceptions lernen.

## Kamingespräche



Heute Abend: **Eine Instanzvariable und eine lokale Variable sprechen (mit erstaunlicher Zurückhaltung) über Leben und Tod.**

**Instanzvariable**

Ich würde gern anfangen, weil ich für Programme meist wichtiger bin als eine lokale Variable. Ich soll Objekte unterstützen. In der Regel über die ganze Lebensdauer des Objekts hinweg. Was soll schon ein Objekt ohne *Zustand* sein? Und was ist ein Zustand? Werte, die mit ***Instanzvariablen*** festgehalten werden.

Nein. Versteh mich nicht falsch. Deine Bedeutung in Methoden ist mir vollkommen klar. Es ist nur, dass dein Leben so kurz ist. So vorübergehend. Deswegen nennt man euch Typen doch »temporäre Variablen«.

Entschuldige vielmals. Das verstehe ich vollkommen.

Darüber habe ich noch nie nachgedacht. Was macht ihr, wenn die anderen Methoden laufen und ihr darauf wartet, dass euer Frame wieder oben auf dem Stack liegt?

**Lokale Variable**

Ich kann deine Sichtweise nachvollziehen. Die Bedeutung von Objektzuständen und all dem ist mir natürlich vollkommen klar. Man sollte die Leute aber nicht in die Irre führen. Lokale Variablen sind *echt* wichtig. Um mal eine deiner Formulierungen zu verwenden: »Was soll schon ein Objekt ohne *Verhalten* sein?« Und was ist Verhalten? Algorithmen in Methoden. Und du kannst deine Bits darauf wetten, dass da ein paar *lokale Variablen* im Spiel sind, die dafür sorgen, dass diese Algorithmen funktionieren.

In der Lokale-Variablen-Gemeinschaft wird die Bezeichnung »temporäre Variablen« als abqualifizierend betrachtet. Wir ziehen »lokale Variable«, »Stack-Variable«, »automatische Variable« oder »geltungseingeschränkte Variable« vor.

Trotzdem ist es richtig, dass wir kein langes Leben haben und dass es auch kein besonders *gutes* Leben ist. Erst werden wir mit all den anderen lokalen Variablen in einen Stack-Frame gepresst ... und wenn dann die Methode, von der wir ein Teil sind, eine andere Methode aufruft, wird auf uns auch noch ein anderer Frame gepackt. Und wenn *diese* Methode ihrerseits wieder eine *andere* Methode aufruft ... und so weiter. Manchmal müssen wir Ewigkeiten warten, bis die Methoden, die auf dem Stack über uns liegen, fertig sind, damit unsere Methode wieder laufen kann.

Nichts. Gar nichts. Das ist, als wäre man in Tiefschlaf – was die Leute in Science-Fiction-Filmen immer machen, wenn sie große Entfernungen überbrücken müssen. Eigentlich eine Art ausgesetztes Leben. Wir sitzen einfach da in der Warteschleife. Solange unser Frame noch da ist, sind

**Instanzvariable**

Ich habe mal einen Aufklärungsfilm darüber gesehen. Sieht nach einem ziemlich grausamen Ende aus. Wenn die Methode endet, weil sie auf die schließende geschweifte Klammer stößt, wird der Frame im Wortsinn vom Stack *geblasen*! *Das* muss schon schmerzhaft sein.

Ich lebe mit den Objekten auf dem Heap. Na, nicht *mit* den Objekten, sondern eigentlich *in* einem Objekt. Dem Objekt, dessen Zustand ich festhalte. Ich muss zugeben, dass das Leben auf dem Heap eine ziemlich luxuriöse Angelegenheit sein kann. Viele von uns fühlen sich schuldig, insbesondere während der Feiertage.

Okay. Theoretisch ja. Wenn ich eine Instanzvariable von Halsband bin und das Halsband von der GC geschluckt wird, werden auch die Instanzvariablen von Halsband weggeworfen wie die Zeitung vom Vortag. Aber mir hat man erzählt, das würde fast nie passieren.

Sie lassen uns *trinken*?

**Lokale Variable**

wir und der Wert, den wir festhalten, sicher. Aber es ist eine zweifelhafte Freude, wenn unser Frame wieder läuft. Einerseits sind wir endlich wieder aktiv. Andererseits nagt die Uhr wieder an unseren kurzen Leben. Je länger unsere Methode läuft, um so näher kommen wir dem Ende der Methode. Und wir wissen *alle*, was dann passiert.

Das musst du mir nicht sagen. Normalerweise verwendet man den Begriff *entfernen*, wie in »der Frame wurde vom Stack entfernt«. Das klingt so sachlich, so nach Hausputz. Aber du hast den Film ja gesehen. Warum reden wir nicht ein bisschen über dich? Ich weiß, wie mein kleiner Stack-Frame aussieht, aber wo lebst *du* eigentlich?

Aber ihr lebt nicht *immer* so lang wie das Objekt, das euch deklariert hat, oder? Nehmen wir beispielsweise an, es gibt ein Hund-Objekt mit einer Halsband-Instanzvariable. Stell dir vor, *du* wärst eine Instanzvariable des *Halsband*-Objekts. Vielleicht eine Referenz auf eine Namensschild-Instanz oder so was. Und du sitzt da glücklich in dem *Halsband*-Objekt, das glücklich in dem *Hund*-Objekt sitzt. Aber ... was passiert, wenn der Hund ein neues Halsband will oder seine Halsband-Instanzvariable auf *null* setzt? Dann kann das Halsband-Objekt für die GC ausgewählt werden. Und wenn *du* eine Instanzvariable in Halsband bist und das ganze *Halsband* aufgegeben wird, was passiert dann mit *dir*?

Und das hast du geglaubt? Das erzählen die uns, damit wir motiviert und produktiv bleiben. Aber vergisst du da nicht etwas? Was ist, wenn du eine Instanzvariable in einem Objekt bist und dieses Objekt *nur* von einer *lokalen* Variablen referenziert wird? Wenn ich die einzige Referenz auf das Objekt bin, in dem du steckst, nehme ich dich mit, wenn ich gehe. Ob es dir gefällt oder nicht, unser Schicksal kann verknüpft sein. Deswegen schlage ich vor, dass wir den ganzen Kram jetzt vergessen und uns besaufen, so lange wir die Zeit dazu haben. Carpe RAM und alles.

# Spielen Sie Garbage Collector

Welche der Codezeilen auf der rechten Seite würde bewirken, dass genau ein weiteres Objekt für den Garbage Collector auswählbar wird, wenn sie am Punkt A eingefügt wird? (Gehen Sie davon aus, dass die Ausführung des Punkts A (// weitere Methoden aufrufen) lange dauert und dem Garbage Collector Zeit gibt, seine Arbeit zu machen.)

```
public class GC {
  public static GC machZeug() {
    GC newGC = new GC();
    machZeug2(newGC);
    return newGC;
  }

  public static void main(String [] args) {
    GC gc1;
    GC gc2 = new GC();
    GC gc3 = new GC();
    GC gc4 = gc3;
    gc1 = machZeug();
    A
    // weitere Methoden aufrufen
  }

  public static void machZeug2(GC gcKopie) {
    GC lokalesGC;
  }
}
```

1 `gcKopie = null;`

2 `gc2 = null;`

3 `newGC = gc3;`

4 `gc1 = null;`

5 `newGC = null;`

6 `gc4 = null;`

7 `gc3 = gc2;`

8 `gc1 = gc4;`

9 `gc3 = null;`

# Beliebte Objekte

In diesem Codebeispiel werden mehrere neue Objekte erzeugt. Sie haben die Aufgabe herauszufinden, welches Objekt das »beliebteste« ist, d.h., auf welches die meisten Referenzvariablen verweisen. Geben Sie dann an, wie *viele* Referenzen es für dieses Objekt insgesamt gibt und welche das sind! Als Hinweis zeigen wir Ihnen eins der neuen Objekte und die Referenzvariable, die es referenziert.

Viel Glück!

```
class Bienen {
  Honig [] bienenHonig;
}

class Waschbär {
  Topf t;
  Honig waschbärHonig;
}

class Topf {
  Honig honig;
}

class Bär {
  Honig bärenHonig;
}

public class Honig {
  public static void main(String [] args) {
    Honig honigTopf = new Honig();
    Honig [] honig = {honigTopf, honigTopf, honigTopf, honigTopf};
    Bienen b1 = new Bienen();
    b1.bienenHonig = honig;
    Bär [] bären = new Bär[5];
    for (int x=0; x < 5; x++) {
      bären[x] = new Bär();
      bären[x].bärenHonig = honigTopf;
    }
    Topf t = new Topf();
    t.honig = honigTopf;
    Waschbär wb = new Waschbär();

    wb.waschbärHonig = honigTopf;
    wb.t = t;
    t = null;
  } // main-Ende
}
```

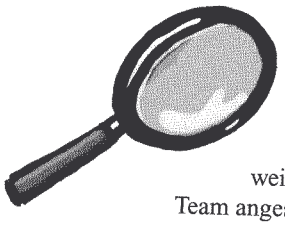
Hier ist ein neues Waschbär-Objekt!

Hier ist seine Referenzvariable »wb«.

## Kurzkrimi

»Wir haben die Simulation viermal laufen lassen, aber die Temperatur des Hauptmoduls wandert immer vom Nominalwert abwärts«, sagte Sarah entnervt. »Letzte Woche haben wir unsere neuen Temperatur-Bots installiert. Die Messwerte der Kühler-Bots, die die Kühlung der Wohnquartiere unterstützen sollen, scheinen der Spec zu entsprechen. Deswegen haben wir unsere Analyse auf die Wärmespeicher-Bots konzentriert, die Bots, die das Heizen der Wohnquartiere unterstützen sollen.« Tim seufzte. Zunächst schien es, dass die Nano-Technologie ihnen einen Vorsprung im Zeitplan verschaffen würde. Aber jetzt, da nur noch fünf Wochen bis zum Start blieben, bestanden einige der wichtigen Lebenserhaltungssysteme der Raumstation den Simulationsspießrutenlauf immer noch nicht.

»Welche Verhältnisquotienten simuliert ihr?«, fragte Tim.



»Wenn ich richtig verstehe, worauf du hinauswillst, dann haben wir das bereits in Betracht gezogen«, erwiderte Sarah. »Mission Control akzeptiert die kritischen Systeme nicht, wenn wir im Test den Rahmen der Spezifikation verlassen. Wir müssen die V3-Kühler-Bot-SimEinheiten in einem 2:1-Verhältnis mit den V2-Kühler-SimEinheiten testen«, fuhr Sarah fort. »Insgesamt soll das Verhältnis von Wärmespeicher-Bots zu Kühler-Bots 4:3 betragen.«

»Wie ist der Energieverbrauch, Sarah?«, fragte Tim. Sarah machte eine Pause. »Das ist ein weiteres Problem. Der Energieverbrauch ist höher als vorhergesehen. Auch darauf haben wir ein Team angesetzt. Aber weil die Nanos drahtlos sind, ist es schwierig, den Energieverbrauch der Kühler-Bots von dem der Wärmespeicher-Bots zu trennen.« »Insgesamt soll der Energieverbrauch«, ergänzte Sarah, »im Verhältnis 3:2 stehen, wobei die Kühler mehr Energie aus dem Wireless Grid ziehen dürfen.«

»Okay, Sarah«, sagte Tim. »Werfen wir einen Blick auf die Initialisierungsteile des Simulationscodes. Wir müssen dieses Problem finden, und wir müssen es schnell finden!«

```
import java.util.*;
class V2Kühler {
  V2Kühler(ArrayList list) {
    for(int x=0; x<5; x++) {
      list.add(new SimEinheit("V2Kühler"));
    }
  }
}

class V3Kühler extends V2Kühler {
  V3Kühler(ArrayList lglist) {
    super(lglist);
    for(int g=0; g<10; g++) {
      lglist.add(new SimEinheit("V3Kühler"));
    }
  }
}

class WärmespeicherBot {
  WärmespeicherBot(ArrayList rlist) {
    rlist.add(new SimEinheit("Wärmespeicher"));
  }
}
```

## Kurzkrimi (Fortsetzung)

```
public class TestLebenserhaltungsSim {
  public static void main(String [] args) {
    ArrayList aList = new ArrayList();
    V2Kühler v2 = new V2Kühler(aList);
    V3Kühler v3 = new V3Kühler(aList);
    for(int z=0; z<20; z++) {
      WärmespeicherBot speicher = new WärmespeicherBot(aList);
    }
  }
}

class SimEinheit {
  String botTyp;
  SimEinheit(String typ) {
    botTyp = typ;
  }
  int energieVerbrauch() {
    if ("Wärmespeicher".equals(botTyp)) {
      return 2;
    } else {
      return 4;
    }
  }
}
```

Tim warf einen kurzen Blick auf den Code, und da erschien ein kleines Lächeln auf seine Lippen. »Ich glaube, ich habe das Problem gefunden, Sarah. Und ich wette, ich weiß auch, um wie viel Prozent deine Energieverbrauchswerte abweichen!«

Welchen Verdacht hatte Tim? Wie konnte er die Abweichung der Energieverbrauchswerte raten? Welche paar Codezeilen würden Sie dem Programm hinzufügen, um den Fehler zu beheben?

## Lösungen zu den Übungen

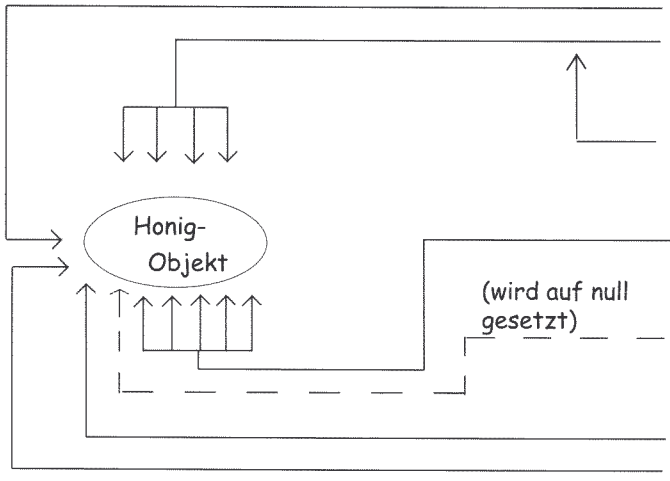
### G.C.

| | | |
|---|---|---|
| 1 | `gcKopie = null;` | Nein - diese Zeile versucht auf eine Variable zuzugreifen, die nicht mehr in Geltung ist. |
| 2 | `gc2 = null;` | Okay - gc2 war die einzige Referenzvariable, die das Objekt referenzierte. |
| 3 | `newGC = gc3;` | Nein - wieder eine Variable, die nicht mehr in Geltung ist. |
| 4 | `gc1 = null;` | Okay - gc1 hielt die einzige Referenz, weil newGC bereits tot ist. |
| 5 | `newGC = null;` | Nein - newGC ist nicht mehr in Geltung. |
| 6 | `gc4 = null;` | Nein - das Objekt wird immer noch von gc3 referenziert. |
| 7 | `gc3 = gc2;` | Nein - das Objekt wird immer noch von gc4 referenziert. |
| 8 | `gc1 = gc4;` | Okay - der einzigen Referenz auf das Objekt wird ein neues Objekt zugewiesen. |
| 9 | `gc3 = null;` | Nein - das Objekt wird immer noch von gc4 referenziert. |

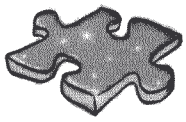
---

### Beliebte Objekte

Wahrscheinlich war es nicht besonders schwer herauszufinden, dass das Honig-Objekt, das zuerst von der Variablen honigTopf referenziert wurde, das bei Weitem »beliebteste« Objekt in dieser Klasse ist. Aber es war vielleicht etwas schwerer zu erkennen, dass alle Variablen aus dem Code, die auf das Honig-Objekt zeigen, ***dasselbe Objekt referenzieren***! Es gibt insgesamt zwölf aktive Referenzen auf das Objekt, bevor die main( )-Methode fertig ist. Die Variable *t.honig* ist eine Zeit lang gültig, aber t wird am Ende auf null gesetzt. Da *wb.t* immer noch auf das Topf-Objekt verweist, referenziert *wb.t.honig* das Objekt immer noch (obwohl es nie explizit deklariert wurde)!




```
public class Honig {
  public static void main(String [] args) {
    Honig honigTopf = new Honig();
    Honig [] honig = {honigTopf, honigTopf,
                      honigTopf, honigTopf};
    Bienen b1 = new Bienen();
    b1.bienenHonig = honig;
    Bär [] bären = new Bär[5];
    for (int x=0; x < 5; x++) {
      bären[x] = new Bär();
      bären[x].bärenHonig = honigTopf;
    }
    Topf t = new Topf();
    t.honig = honigTopf;
    Waschbär wb = new Waschbär();

    wb.waschbärHonig = honigTopf;
    wb.t = t;
    t = null;
  } // main-Ende
}
```

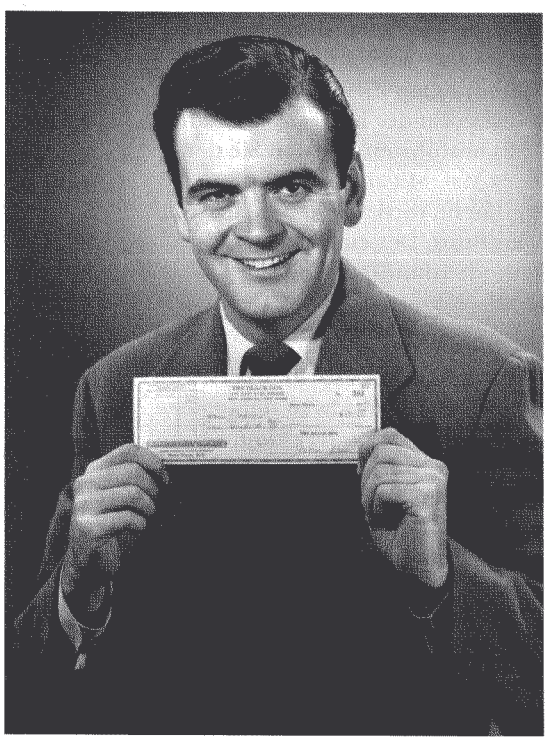


## Kurzkrimi-Lösung

Tim bemerkte, dass der Konstruktor für die Klasse V2Kühler eine ArrayList erwartete. Das bedeutete, dass jedes Mal, wenn der *V3*Kühler-Konstruktor aufgerufen wurde, im super()-Aufruf an den V2Kühler-Konstruktor eine ArrayList übergeben wurde. Das hieß, dass jedes Mal fünf zusätzliche V2Kühler-SimEinheiten erzeugt wurden. Wenn Tim Recht hatte, würde der Gesamtenergieverbrauch 120 betragen und nicht die 100, die Sarahs angenommene Verhältnisquotienten erwarten ließen.

Da alle Bot-Klassen SimEinheiten erstellen, hätte man das Problem leicht herausfinden können, indem man einen SimEinheit-Konstruktor geschrieben hätte, der jedes Mal eine Zeile ausgibt, wenn eine SimEinheit erzeugt wird!

# Zahlen, bitte!



**Rechnen Sie's aus.** Aber zum Arbeiten mit Zahlen gehört noch mehr als nur einfache Arithmetik. Sie möchten vielleicht den absoluten Betrag einer Zahl erhalten, eine Zahl runden oder die größere von zwei Zahlen ermitteln. Vielleicht möchten Sie auch, dass Ihre Zahlen mit exakt zwei Nachkommastellen ausgegeben oder dass große Zahlen mit Trennzeichen unterteilt werden, damit man sie leichter lesen kann. Und wie steht es mit Datumsangaben? Vielleicht möchten Sie Datumsangaben auf unterschiedliche Art ausgeben oder sogar Datumsangaben *manipulieren*, z.B. so: »Addiere drei Wochen zum heutigen Datum.« Und wie ist es mit der Überführung eines Strings in eine Zahl? Oder mit der Umwandlung einer Zahl in einen String? Sie haben Glück. In der Java-API gibt es eine Menge praktischer und leicht anwendbarer »Zahlenverdreher«-Methoden. Allerdings sind die meisten davon **statisch**, deshalb sehen wir uns zunächst einmal an, was es für eine Variable oder eine Methode bedeutet, statisch zu sein – einschließlich der Konstanten in Java, der statischen *finalen* Variablen.

# MATH-Methoden: das *Globalste*, was Java an Methoden zu bieten hat

Abgesehen davon, dass es in Java natürlich *überhaupt nichts* Globales gibt. Aber überlegen Sie doch einmal Folgendes: Was ist mit einer Methode, deren Verhalten nicht vom Wert einer Instanzvariablen abhängt? Nehmen Sie beispielsweise die Methode round() in der Klasse Math. Sie tut jedes Mal das Gleiche – sie rundet eine Gleitkommazahl (das Argument der Methode) auf die nächste ganze Zahl. Jedes Mal. Wenn Sie 10.000 Instanzen der Klasse Math hätten und damit die Methode round(42.2) ausführten, würden Sie den ganzzahligen Wert 42 erhalten. Jedes Mal. Mit anderen Worten, die Methode macht etwas mit dem Argument, wird aber dabei nie vom Zustand einer Instanzvariablen beeinflusst. Der einzige Wert, der Einfluss darauf nimmt, wie die Methode round() abläuft, ist das der Methode übergebene Argument!

Sieht das nicht nach einer Verschwendung von kostbarem Platz auf dem Heap aus, wenn man eine Instanz der Klasse Math erstellt, nur um die round()-Methode auszuführen? Und was ist mit *anderen* Math-Methoden wie min(), der man zwei numerische primitive Werte übergibt, von denen sie den kleineren zurückgibt? Oder max()? Oder abs() – eine Methode, die den absoluten Betrag einer Zahl zurückgibt?

***Diese Methoden benutzen niemals Werte von Instanzvariablen.*** Tatsächlich *besitzt* die Klasse Math gar keine Instanzvariablen. Durch die Erzeugung einer Instanz der Klasse Math hätte man also nichts gewonnen. Also, was glauben Sie? Genau – Sie müssen keine Instanz erzeugen! Tatsache ist: Es geht gar nicht.

**Die Methoden in der Klasse Math greifen nicht auf Werte von Instanzvariablen zurück. Und weil diese Methoden »statisch« sind, brauchen Sie auch keine Instanz von Math. Alles, was Sie brauchen, ist die Klasse Math.**

```
int x = Math.round(42.2);
int y = Math.min(56,12);
int z = Math.abs(-343);
```

Diese Methoden benutzen keinerlei Instanzvariablen, ihr Verhalten hängt daher nicht von Informationen über ein bestimmtes Objekt ab.

### Wenn Sie versuchen, eine Instanz der Klasse Math zu erzeugen:

```
Math mathObject = new Math();
```

### ... erhalten Sie folgende Fehlermeldung:

```
Datei Bearbeiten Fenster Hilfe IchDachteEsGäbeHierKeineMathematik
% javac TestMath.java
TestMath.java:3: Math() has private
access in java.lang.Math
  Math mathObject = new Math();
                    ^
1 error
```

Diese Fehlermeldung zeigt, dass der Math-Konstruktor privat ist! Das bedeutet: Sie können NIEMALS mit Hilfe von >>new<< aus der Klasse Math ein neues Math-Objekt erzeugen.

# Der Unterschied zwischen regulären (nicht-statischen) und statischen Methoden

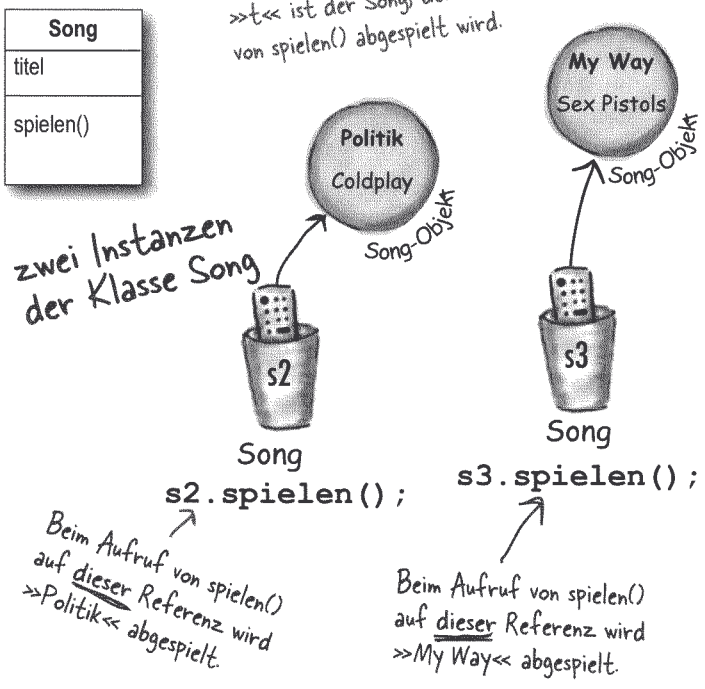
Java ist objektorientiert, aber ab und zu gibt es Spezialfälle, typischerweise Hilfsmethoden (wie die Math-Methoden), für die keine Instanz der Klasse benötigt wird. Mit dem Schlüsselwort **`static`** kann eine Methode auch ***ohne irgendeine Instanz der Klasse*** ausgeführt werden. »Statische Methode« bedeutet: »Verhalten nicht von einer Instanzvariablen abhängig, daher keine Instanz/kein Objekt erforderlich. Nur die Klasse.«

## reguläre (nicht-statische) Methode

```
public class Song {
  String titel;
  public Song(String t) {
    titel = t;
  }
  public void spielen {
    SoundPlayer player = new SoundPlayer();
    player.soundAbspielen(titel);
  }
}
```

Der Wert der Instanzvariablen wirkt sich auf das Verhalten der Methode spielen() aus.

Der aktuelle Wert der Instanzvariablen >>t<< ist der Song, der beim Aufruf von spielen() abgespielt wird.

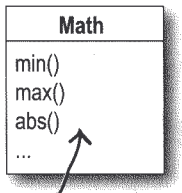



Beim Aufruf von spielen() auf dieser Referenz wird >>Politik<< abgespielt.

Beim Aufruf von spielen() auf dieser Referenz wird >>My Way<< abgespielt.

## statische Methode

```
public static int min(int a, int b){
  // gibt von a und b den niedrigeren
  // Wert zurück
}
```



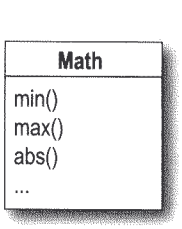

Keine Instanzvariablen. Das Verhalten der Methode ändert sich nicht mit dem Zustand von Instanzvariablen.

```
Math.min(42,36);
```

Verwenden Sie den Klassennamen an Stelle des Namens einer Referenzvariablen.

KEINE OBJEKTE!! Es kommen absolut KEINE OBJEKTE in diesem Bild vor!

**Rufen Sie eine statische Methode über den Namen einer Klasse auf**




```
Math.min(88,86);
```

**Rufen Sie eine nicht-statische Methode über den Namen einer Referenzvariablen auf**

```
Song t2 = new Song();
t2.spielen();
```

# Was es bedeutet, eine Klasse mit statischen Methoden zu haben.

Häufig (aber nicht immer) soll eine Klasse mit statischen Methoden nicht instantiiert werden. In Kapitel 8 haben wir über abstrakte Klassen gesprochen und darüber, dass man bei einer Klasse, die mit dem Modifizierer **`abstract`** markiert ist, niemals »new« schreiben kann. Mit anderen Worten: ***Es ist unmöglich, eine abstrakte Klasse zu instantiieren.***

Sie können die Instantiierung einer *nicht*-abstrakten Klasse durch fremden Code unterbinden, indem Sie den Konstruktor als **`private`** kennzeichnen. Wie Sie wissen, kann eine *Methode*, die als private markiert ist, ausschließlich durch Code innerhalb dieser Klasse aufgerufen werden. Bei einem als privat gekennzeichneten *Konstruktor* ist es im Wesentlichen das Gleiche – der Konstruktor kann nur durch Code innerhalb der Klasse aufgerufen werden. Niemand kann von *außerhalb* der Klasse »new« sagen. So funktioniert es beispielsweise bei der Klasse Math. Der Konstruktor ist privat, daher können Sie keine Instanz der Klasse Math erzeugen. Der Compiler weiß, dass Ihr Code keinen Zugriff auf diesen privaten Konstruktor hat.

Das bedeutet allerdings nicht, dass eine Klasse mit einer oder mehreren statischen Methoden nie instantiiert werden sollte. Tatsächlich enthält ja jede Klasse, der Sie eine main()-Methode mitgeben, eine statische Methode!

Typischerweise schreiben Sie eine main()-Methode, um eine andere Klasse zu starten oder zu testen, und fast immer instantiieren Sie dabei in main() eine Klasse und rufen dann eine Methode auf dieser neuen Instanz auf.

Sie dürfen also statische und nicht-statische Methoden in einer Klasse kombinieren; wenn allerdings auch nur eine einzige nicht-statische Methode vorkommt, bedeutet das, dass man auf *irgendeine* Weise eine Instanz der Klasse erzeugen können muss. Der einzige Weg, an ein neues Objekt zu gelangen, führt über »new« oder über die später noch zu besprechende Deserialisierung (oder über etwas, das Java Reflection API heißt und von uns nicht besprochen wird). Es gibt keine andere Möglichkeit. Aber *wer* genau da »new« sagt, kann eine interessante Frage sein, mit der wir uns etwas weiter unten in diesem Kapitel noch beschäftigen werden.

# Statische Methoden können keine nicht-statischen (Instanz-)Variablen benutzen!

Statische Methoden haben während ihrer Ausführung keine Kenntnis von irgendeiner Instanz ihrer Klasse. Und wie Sie auf den vorhergehenden Seiten gesehen haben, gibt es vielleicht auch gar keine Instanzen der Klasse. Da eine statische Methode mit Hilfe der *Klasse* aufgerufen wird (*Math*.random()) und nicht über eine *Instanzreferenz* (*t2*.spielen()), kann sich eine statische Methode nicht auf irgendwelche Instanzvariablen der Klasse beziehen. Die statische Methode weiß ja nicht, den Wert *welcher* Instanz sie nehmen soll.

## Wenn Sie versuchen, den folgenden Code zu kompilieren ...

```
public class Ente {

  private int größe;

  public static void main (String[] args) {
    System.out.println("Größe ist " + größe);
  }

  public void setGröße(int g) {
    größe = g;
  }

  public int getGröße() {
    return größe;
  }
}
```

Welche Ente? Wessen Größe?

Wenn es irgendwo auf dem Heap eine Ente geben sollte, kennen wir sie jedenfalls nicht.

## ... erhalten Sie diese Fehlermeldung:

```
Datei Bearbeiten Fenster Hilfe Quak
% javac Ente.java
Ente.java:6: non-static variable größe cannot be referenced from a static context
        System.out.println("Größe ist " + größe);
                                          ^
```

Wenn Sie versuchen, eine Instanzvariable aus einer statischen Methode heraus zu benutzen, denkt der Compiler: »Die Instanzvariable von welchem Objekt? Ich weiß nicht, *welches* Sie meinen!« Auch wenn Sie zehn Enten-Objekte auf dem Heap haben - ein statisches Member kennt keins davon.

# Statische Methoden können auch keine nicht-statischen Methoden benutzen!

Was machen denn nicht-statische Methoden? ***Normalerweise hängt ihr Verhalten vom Zustand der Instanzvariablen ab.*** Eine Methode getName() gibt den Wert der Variablen name zurück. Und zwar den Namen des Objekts, das für den Aufruf der Methode getName() benutzt wird.

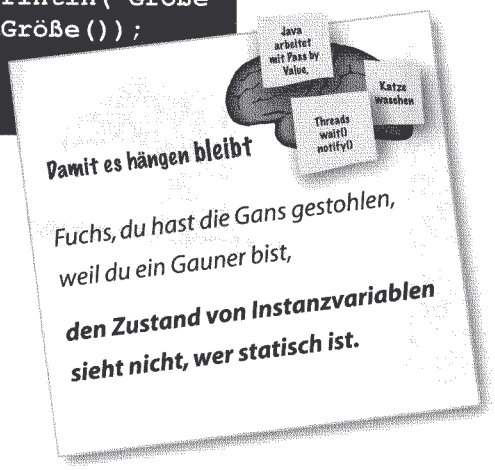
## Das hier ist nicht kompilierbar:

```
public class Ente {

  private int größe;

  public static void main(String[] args) {
    System.out.println("Größe der Ente ist " + getGröße());
  }

  public void setGröße(int g) {
    größe = g;
  }

  public int getGröße() {
    return größe;
  }
}
```

Der Aufruf von getGröße() schiebt das Unvermeidliche nur hinaus – getGröße() verwendet die Instanzvariable größe.

Wieder das gleiche Problem ... wessen Größe?

```
Datei Bearbeiten Fenster Hilfe GibsAuf
% javac Ente.java
Ente.java:6: non-static method
getGröße() cannot be referenced
from a static context
        System.out.println("Größe
der Ente ist " + getGröße());
                 ^
```




## Es gibt keine Dummen Fragen

**F: Was ist, wenn Sie versuchen, eine nicht-statische Methode aus einer statischen Methode heraus aufzurufen, aber die nicht-statische Methode keinerlei Instanzvariablen benutzt? Erlaubt der Compiler das?**

A: Nein. Der Compiler weiß, dass Sie Instanzvariablen in einer nicht-statischen Methode benutzen *können* – egal, ob Sie es tun oder nicht. Und denken Sie mal an die Folgen ... Stellen Sie sich vor, Sie dürften ein solches Szenario kompilieren. Was passiert dann später, wenn Sie die Implementierung dieser nicht-statischen Methode eines Tages so ändern wollen, dass sie nun doch auf eine Instanzvariable zurückgreift? Oder – noch schlimmer – was passiert, wenn eine Unterklasse die Methode *überschreibt* und in der überschreibenden Version eine Instanzvariable benutzt wird?

**F: Ich könnte schwören, ich habe schon Code gesehen, in dem eine statische Methode über eine Referenzvariable und nicht über den Klassennamen aufgerufen wird.**

A: Das *kann* man so machen. Aber wie sagte auch Ihre Mutter bestimmt immer zu Ihnen: »Nur weil etwas erlaubt ist, ist es noch lange nicht gut.« Es *funktioniert* zwar, eine statische Methode über eine beliebige Instanz der Klasse aufzurufen, aber man erhält dadurch irreführenden (d.h. schlechter lesbaren) Code. Sie *dürfen* schreiben:

```
Ente e = new Ente();
String[] s = {};
e.main(s);
```

Dieser Code ist erlaubt, aber der Compiler löst das sowieso wieder zur echten Klasse auf. (»Okay, *e* ist vom Typ Ente, und main() ist statisch, also rufe ich die statische Methode main() in der Klasse Ente auf.«) Anders ausgedrückt: Wenn man *e* für den Aufruf von main() benutzt, folgt daraus nicht, dass main() irgendeine besondere Kenntnis des von *e* referenzierten Objekts hat. Es ist lediglich ein alternativer Weg für den Aufruf einer statischen Methode, aber diese ist nach wie vor statisch!

# Statische Variablen haben für ALLE Instanzen der Klasse den gleichen Wert

Stellen Sie sich vor, Sie wollten zählen, wie viele Ente-Instanzen erzeugt werden, während Ihr Programm läuft. Wie würden Sie das machen? Vielleicht mit einer Instanzvariablen, die Sie im Konstruktor hochzählen?

```
class Ente {
  int entenAnzahl = 0;
  public Ente() {
    entenAnzahl++;
  }
}
```

Dadurch würde entenAnzahl jedes Mal, wenn eine Ente erzeugt wird, auf 1 gesetzt.

Nein, das würde nicht funktionieren, weil entenAnzahl eine Instanzvariable ist und bei jeder Ente mit 0 beginnt. Sie könnten versuchen, eine Methode in irgendeiner anderen Klasse aufzurufen, aber das ist Pfusch. Sie brauchen eine Klasse mit nur einem einzigen Exemplar der Variablen, die von allen Instanzen benutzt wird.

Und genau das bietet Ihnen eine statische Variable: einen Wert, den alle Instanzen einer Klasse miteinander teilen – also einen Wert pro *Klasse* anstatt eines Werts pro *Instanz*.

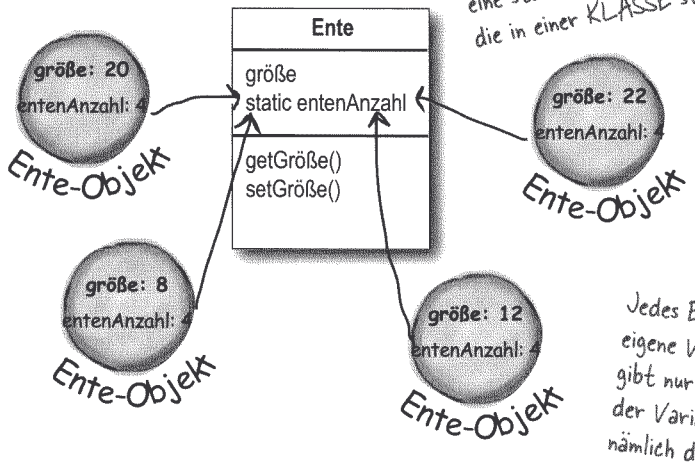
```
public class Ente {

  private int größe;
  private static int entenAnzahl = 0;

  public Ente() {
    entenAnzahl++;
  }

  public void setGröße(int g) {
    größe = g;
  }

  public int getGröße() {
    return größe;
  }
}
```

Die statische Variable entenAnzahl wird NUR initialisiert, wenn die Klasse geladen wird, aber NICHT bei jedem Mal, wenn eine neue Instanz erzeugt wird.

Jetzt wird sie jedes Mal, wenn der Entenkonstruktor ausgeführt wird, weiter inkrementiert, weil entenAnzahl statisch ist und nicht auf 0 zurückgesetzt wird.

Ein Ente-Objekt enthält keine eigene Kopie von entenAnzahl.

Weil entenAnzahl statisch ist, teilen sich alle Ente-Objekte ein einziges Exemplar davon. Sie können sich eine statische Variable als eine Variable vorstellen, die in einer KLASSE statt in einem Objekt lebt.




Jedes Ente-Objekt hat seine eigene Variable größe, aber es gibt nur ein einziges Exemplar der Variablen entenAnzahl – nämlich das in der Klasse.

## Statische Variablen werden gemeinsam genutzt.

## Alle Instanzen der gleichen Klasse teilen sich ein einziges Exemplar der statischen Variablen.

Instanzvariablen: eine pro **Instanz**

statische Variablen: eine pro **Klasse**

## Gehirntraining

Wie wir weiter oben in diesem Kapitel gesehen haben, bedeutet ein privater Konstruktor, dass man die Klasse nicht aus Code heraus instantiieren kann, der außerhalb der Klasse läuft. Mit anderen Worten: Nur Code innerhalb der Klasse kann bei einer Klasse mit einem privaten Konstruktor eine neue Instanz dieser Klasse erzeugen. (Hier haben wir eine Art von Henne-oder-Ei-Problem.)

Wie könnten Sie eine Klasse so schreiben, dass von ihr nur EINE Instanz erzeugt werden kann und alle, die eine Instanz der Klasse benutzen wollen, ausschließlich diese eine, einzige Instanz benutzen?

# Die Initialisierung einer statischen Variablen

Statische Variablen werden initialisiert, wenn *eine Klasse geladen wird*. Eine Klasse wird geladen, weil die JVM entscheidet, dass es Zeit dafür ist. Im typischen Fall lädt die JVM eine Klasse, weil jemand versucht, erstmalig eine Instanz der Klasse zu erzeugen oder eine statische Methode oder Variable der Klasse zu benutzen. Als Programmierer können Sie die JVM auch direkt anweisen, die Klasse zu laden, aber davon werden Sie wahrscheinlich nie Gebrauch machen müssen. Fast immer fahren Sie besser damit, wenn Sie die JVM entscheiden lassen, wann eine Klasse *geladen* wird.

**Alle statischen Variablen in einer Klasse werden initialisiert, *bevor* irgendein Objekt dieser Klasse erzeugt werden kann.**

Und bei statischer Initialisierung werden Ihnen zwei Dinge garantiert:

Statische Variablen in einer Klasse werden initialisiert, bevor irgendein *Objekt* dieser Klasse erzeugt werden kann.

Statische Variablen in einer Klasse werden initialisiert, bevor irgendeine *statische Methode* der Klasse ausgeführt wird.

```
class Spieler {
  static int spielerAnzahl = 0;
  private String name;
  public Spieler(String n) {
    name = n;
    spielerAnzahl++;
  }
}

public class SpielerTestlauf {
  public static void main (String[] args) {
    System.out.println(Spieler.spielerAnzahl);
    Spieler eins = new Spieler("Tiger Woods");
    System.out.println(Spieler.spielerAnzahl);
  }
}
```

spielerAnzahl wird beim Laden der Klasse initialisiert. Wir haben sie explizit mit 0 initialisiert, aber eigentlich ist das nicht nötig, weil 0 der Standardwert für int-Variablen ist. Genau wie Instanzvariablen erhalten auch statische Variablen Standardwerte.

Die Standardwerte für deklarierte, aber nicht initialisierte statische und Instanzvariablen sind die gleichen:

elementare Ganzzahlen (long, short usw.): 0
elementare Gleitkommazahlen (float, double): 0.0
Boolesche Variablen: false
Objektreferenzen: null

Greifen Sie auf eine statische Variable so zu wie auf eine statische Methode – über den Klassennamen.

Statische Variablen werden initalisiert, wenn die Klasse geladen wird. Initialisieren Sie eine statische Variable nicht ausdrücklich (indem Sie ihr schon bei der Deklaration einen Wert zuweisen), erhält sie einen Standardwert. Variablen vom Typ int werden mit »0« initialisiert; wir brauchen also nicht ausdrücklich »spielerAnzahl = 0« hinzuschreiben. Wenn eine statische Variable deklariert, aber nicht initialisiert wird, erhält sie den Standardwert für diesen Variablentyp – genau wie auch Instanzvariablen bei der Deklaration Standardwerte erhalten.

```
Datei Bearbeiten Fenster Hilfe Was?
% java SpielerTestlauf
0   <- bevor Instanzen erzeugt werden
1   <- nachdem ein Objekt erzeugt wurde
```

# Statische finale Variablen sind Konstanten

Wenn eine Variable als **final** gekennzeichnet ist, heißt das, sie kann sich niemals ändern, wenn sie erst einmal initialisiert ist. Anders ausgedrückt: Der Wert der statischen finalen Variablen wird derselbe bleiben, so lange die Klasse geladen ist. Schlagen Sie einmal Math.PI in der API-Dokumentation nach, dann finden Sie:

```
public static final double PI = 3.141592653589793;
```

Die Variable ist als **public** markiert, so dass jeder Code darauf zugreifen kann.

Die Variable ist als **static** markiert, so dass Sie keine Instanz der Klasse Math brauchen (die Sie, wie Sie sich erinnern werden, auch gar nicht erzeugen dürfen).

Die Variable ist als **final** markiert, weil sich PI nicht ändert (soweit es Java betrifft).

Es gibt keine andere Möglichkeit, eine Variable als konstant zu kennzeichnen. Es gibt jedoch eine Namenskonvention, die Ihnen hilft, konstante Variablen zu erkennen:

***Die Namen konstanter Variablen sollten durchgängig in Großbuchstaben geschrieben werden!***

**Ein statischer Initialisierer ist ein Codeblock, der beim Laden einer Klasse ausgeführt wird, bevor anderer Code diese Klasse benutzen kann; er eignet sich deshalb wunderbar für die Initialisierung einer statischen finalen Variablen.**

```
class Foo {
  final static int x;
  static {
    x = 42;
  }
}
```

## Initialisieren Sie eine *finale* statische Variable ...

**1 bei der Deklaration:**

```
public class Foo {
   public static final int FOO_X = 25;
}
```

Beachten Sie die Namenskonvention – statische finale Variablen sind Konstanten, daher sollte der Name durchgängig in Großbuchstaben geschrieben werden, mit einem Unterstrich zur Trennung der Wörter.

**ODER**

**2 in einem statischen Initialisierer:**

```
public class Bar {
  public static final double BAR_HOCKER;

  static {
    BAR_HOCKER = (double) Math.random();
  }
}
```

Dieser Code wird schon beim Laden der Klasse ausgeführt, noch bevor statische Methoden aufgerufen werden und sogar bevor statische Variablen benutzt werden können.

**Wenn Sie einer finalen Variablen nicht bei einer dieser beiden Gelegenheiten einen Wert zuweisen ...**

```
public class Bar {
  public static final double BAR_HOCKER;
}
```

Keine Initialisierung!

**... fängt der Compiler es ab:**

```
Datei Bearbeiten Fenster Hilfe SoNicht
% javac Bar.java
Bar.java:1: variable BAR_HOCKER
might not have been initialized
1 error
```

# final ist nicht bloß was für statische Variablen ...

Sie können mit dem Schlüsselwort **final** auch nicht-statische Variablen modifizieren, einschließlich Instanzvariablen, lokaler Variablen und sogar Methodenparametern. Das bedeutet in allen Fällen das Gleiche: Der Wert kann nicht geändert werden. Sie können final aber auch verwenden, wenn Sie andere daran hindern wollen, eine Methode zu überschreiben oder eine Unterklasse zu bilden.

## nicht-statische finale Variablen

```
class Fusel {
  final int größe = 3;
  final int whuffie;

  Fusel() {
    whuffie = 42;
  }

  void tuWas(final int x) {
    // Sie können x nicht ändern
  }

  void tuNochWas() {
    final int z = 7;
    // Sie können z nicht ändern
  }
}
```

← Jetzt können Sie größe nicht mehr ändern.

← Jetzt können Sie whuffie nicht mehr ändern.

## finale Methode

```
class Wusel {
  final void berechneWhuffie() {
    // wichtige Sachen, die niemals
    // überschrieben werden dürfen
  }
}
```

## finale Klasse

```
final class MeineAbsolutPerfekteKlasse {
  // kann nicht erweitert werden
}
```

**Wenn eine Variable final ist, können Sie ihren Wert nicht ändern.**

**Wenn eine Methode final ist, können Sie sie nicht überschreiben.**

**Wenn eine Klasse final ist, können Sie sie nicht erweitern (d.h., Sie können keine Unterklasse bilden).**

## Es gibt keine Dummen Fragen

**F: Eine statische Methode kann nicht auf eine nicht-statische Variable zugreifen. Aber kann eine nicht-statische Methode auf eine statische Variable zugreifen?**

A: Natürlich. Eine nicht-statische Methode in einer Klasse kann jederzeit eine statische Methode in dieser Klasse aufrufen oder auf eine statische Variable der Klasse zugreifen.

**F: Warum sollte ich eine Klasse final machen wollen? Läuft das nicht dem Zweck der OO zuwider?**

A: Ja und nein. Ein typischer Grund, eine Klasse final zu machen, ist Sicherheit. Sie können beispielsweise keine Unterklasse der Klasse String schreiben. Stellen Sie sich vor, welchen Schaden jemand anrichten könnte, der die Klasse String erweitern und polymorph dort, wo String-Objekte erwartet werden, seine eigenen String-Unterklassenobjekte einsetzen würde. Wenn Sie sich auf eine bestimmte Implementierung der Methoden in einer Klasse verlassen können müssen, machen Sie die Klasse final.

**F: Ist es nicht überflüssig, die Methoden als final zu kennzeichnen, wenn die Klasse final ist?**

A: Wenn die Klasse final ist, müssen Sie die Methoden nicht als final markieren. Überlegen Sie – wenn eine Klasse final ist, kann sie nie erweitert werden, daher kann keine der Methoden je überschrieben werden.

Andererseits: Wenn Sie anderen erlauben wollen, Ihre Klasse zu erweitern, und Sie möchten, dass einige, aber nicht alle Methoden überschrieben werden können, dann kennzeichnen Sie nicht die ganze Klasse als final, sondern nur selektiv bestimmte Methoden innerhalb der Klasse. Ist eine Methode final, heißt das, dass eine Unterklasse diese spezielle Methode nicht überschreiben kann.

## Punkt für Punkt

- Eine ***statische Methode*** sollte über den Klassennamen statt über eine Objekt-Referenzvariable aufgerufen werden:

```
Math.random() vs. meinFoo.tuWas()
```

- Eine statische Methode kann aufgerufen werden, ohne dass eine Instanz der entsprechenden Klasse auf dem Heap vorhanden ist.
- Eine statische Methode eignet sich gut als Dienstmethode, die nicht vom Wert einer Instanzvariablen abhängig ist (und es nie sein wird).
- Eine statische Methode ist nicht mit einer bestimmten Instanz verknüpft, sondern nur mit der Klasse; sie hat daher keinen Zugriff auf die Werte von Instanzvariablen ihrer Klasse. Sie wüsste ja nicht, von *welcher* Instanz sie die Werte nehmen soll.
- Eine statische Methode kann nicht auf eine nicht-statische Methode zugreifen, da bei nicht-statischen Methoden in der Regel eine Abhängigkeit vom Zustand von Instanzvariablen besteht.
- Enthält Ihre Klasse ausschließlich statische Methoden und Sie möchten nicht, dass diese Klasse instantiiert wird, können Sie den Konstruktor als privat kennzeichnen.
- Eine ***statische Variable*** ist eine Variable, die von allen Membern einer Klasse gemeinsam benutzt wird. Es gibt nur ein einziges Exemplar einer statischen Variablen innerhalb ihrer Klasse und nicht – wie bei Instanzvariablen – eine Kopie pro Instanz.
- Eine statische Methode kann auf eine statische Variable zugreifen.
- Um in Java eine Konstante zu erhalten, kennzeichnen Sie eine Variable sowohl als statisch als auch als final.
- Einer finalen statischen Variablen muss ihr Wert entweder bei der Deklaration zugewiesen werden oder in einem statischen Initialisierer:

```
static {
  HUND_CODE = 420;
}
```

- Gemäß der Namenskonvention für Konstanten (finale statische Variablen) schreibt man deren Namen durchgängig in Großbuchstaben.
- Wenn der Wert einer finalen Variablen einmal zugewiesen wurde, kann er nicht mehr geändert werden.
- Der Wert einer finalen *Instanz*variablen muss entweder bei der Deklaration oder im Konstruktor zugewiesen werden.
- Eine finale Methode kann nicht überschrieben werden.
- Eine finale Klasse kann nicht durch Unterklassen erweitert werden.

Spitzen Sie Ihren Bleistift

## Was ist erlaubt?

Welches der folgenden Beispiele ließe sich – in Anbetracht dessen, was Sie gerade über static und final gelernt haben – kompilieren?

1
```
public class Foo {
   static int x;

   public void los() {
      System.out.println(x);
   }
}
```

2
```
public class Foo2 {
  int x;

  public static void los() {
      System.out.println(x);
  }
}
```

3
```
public class Foo3 {
  final int x;

  public void los() {
      System.out.println(x);
  }
}
```

4
```
public class Foo4 {
  static final int x = 12;

  public void los() {
      System.out.println(x);
  }
}
```

5
```
public class Foo5 {
  static final int x = 12;

  public void los(final int x) {
      System.out.println(x);
  }
}
```

6
```
public class Foo6 {
  int x = 12;

  public static void los(final int x) {
      System.out.println(x);
  }
}
```

# Math-Methoden

Da wir nun wissen, wie statische Methoden funktionieren, wollen wir uns jetzt ein paar statische Methoden in der Klasse Math ansehen. Dies sind nicht alle, nur die Highlights. Sehen Sie sich den Rest in Ihrer API-Dokumentation an, z.B. sqrt(), tan(), ceil(), floor() und asin().

## Math.random()

Gibt einen double-Wert zurück, der größer oder gleich 0.0 und kleiner als 1.0 ist.

```
double r1 = Math.random();
int r2 = (int) (Math.random() * 5);
```

## Math.abs()

Gibt ein double zurück, das gleich dem Betrag des Arguments ist. Die Methode ist überladen - wenn Sie ihr ein int übergeben, gibt sie Ihnen ein int zurück. Übergeben Sie ihr ein double, bekommen Sie ein double.

```
int x = Math.abs(-240);        // gibt 240 zurück
double d = Math.abs(240.40);   // gibt 240.45 zurück
```

## Math.round()

Gibt ein int oder ein long zurück (abhängig davon, ob das Argument ein float oder ein double ist), gerundet auf den nächstliegenden ganzzahligen Wert.

```
int x = Math.round(-24.8f);    //gibt -25 zurück
int y = Math.round(24.45f);    //gibt 24 zurück
```

↑Erinnern Sie sich: Falls Sie kein >>f<< anfügen, wird bei Gleitkommaliteralen angenommen, dass sie vom Typ double sind.

## Math.min()

Der Rückgabewert ist das kleinere der beiden Argumente. Die Methode ist überladen; sie kann int-, long-, float- oder double-Werte annehmen.

```
int x = Math.min(24,240); // gibt 24 zurück
double y = Math.min(90876.5, 90876.49);  // gibt 90876.49
                                         // zurück
```

## Math.max()

Der Rückgabewert ist das größere der beiden Argumente. Die Methode ist überladen; sie kann int-, long-, float- oder double-Werte annehmen.

```
int x = Math.max(24,240); // gibt 240 zurück
double y = Math.max(90876.5, 90876.49);  // gibt 90876.5
                                         // zurück
```

# Einen primitiven Wert einpacken

Manchmal wollen Sie etwas Primitives behandeln wie ein Objekt. In allen Java-Versionen vor 5.0 können Sie beispielsweise primitive Werte nicht direkt in eine Collection wie eine ArrayList oder eine HashMap setzen:

```
int x = 32;
ArrayList list = new ArrayList();
list.add(x);
```

Das funktioniert nicht, außer wenn Sie Java 5.0 oder höher verwenden!! Es gibt keine Methode add(int) in ArrayList, die ein int annimmt! (ArrayList besitzt nur add()-Methoden, die Objektreferenzen erwarten, keine primitiven Werte.)

Für jeden primitiven Typ gibt es eine Wrapper-Klasse, d.h. eine Klasse, in die man ihn »einwickeln« kann. Da die Wrapper-Klassen zum Paket java.lang gehören, müssen sie nicht importiert werden. Sie erkennen Wrapper-Klassen sofort, denn sie sind jeweils nach dem von ihnen umhüllten primitiven Typ benannt; allerdings beginnen sie gemäß der Namenskonvention für Klassen mit einem Großbuchstaben.

Aus irgendwelchen Gründen, über die sich eigentlich keiner auf diesem Planeten so ganz im Klaren ist, haben die API-Designer allerdings beschlossen, dass die Namen der Klassen nicht *ganz genau* den Namen der primitiven Typen entsprechen. Hier sehen Sie, was wir meinen:

**Boolean**
**Character**
**Byte**
**Short**
**Integer**
**Long**
**Float**
**Double**

Aufgepasst! Diese Namen entsprechen nicht genau den primitiven Typen. Für die Klassennamen benutzt man die Langform.

## Das *Ein*packen eines Werts in einen Wrapper

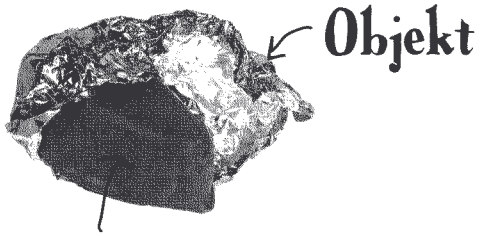
Übergeben Sie den primitiven Wert dem Wrapper-Konstruktor. Das war's schon.

```
int i = 288;
Integer iEingepackt = new Integer(i);
```

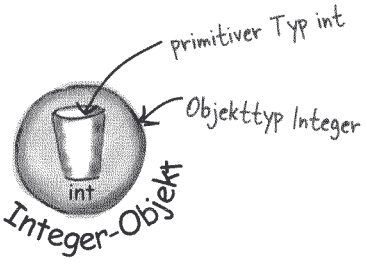
So funktionieren alle Wrapper. Boolean hat eine Methode booleanValue(), Character eine Methode charValue() usw.

## Das *Aus*packen eines Werts

```
int ausgepackt = iEingepackt.intValue();
```




**Wenn Sie etwas Primitives wie ein Objekt behandeln müssen, packen Sie es in einen Wrapper ein. In den Java-Versionen vor 5.0 werden Sie das brauchen, um einen primitiven Wert in einer Collection zu speichern (d.h. in einer Datenstruktur, die eine Ansammlung von anderen Dingen enthält, wie z.B. eine ArrayList oder eine HashMap).**




Anmerkung: Das Bild oben auf der Seite ist ein Stück Schokolade in Silberfolie. Alles klar? Eingewickelt – also mit Wrapper! Manche meinen, das sähe wie eine gegrillte Kartoffel aus – aber das würde auch passen.

Das ist doch dämlich. Sie meinen, ich kann nicht einfach eine ArrayList mit int-Werten machen??? Ich muss jeden einzelnen bescheuerten Wert erst in ein neues Integer-Objekt einpacken und dann wieder auspacken, wenn ich auf den Wert in der ArrayList zugreifen will? Was für eine Zeitverschwendung! Und da sind Fehler ja fast unvermeidlich ...

# Vor Java 5.0 mussten Sie die Arbeit selbst tun ...

Sie hat Recht. Primitives war in allen Java-Versionen vor 5.0 primitiv, und Objektreferenzen waren Objektreferenzen, und sie wurden NIEMALS als austauschbar behandelt. Sie mussten als Programmiererin immer selbst für das Ein- und Auspacken sorgen. Es gab keine Möglichkeit, einen primitiven Wert an eine Methode zu übergeben, die eine Objektreferenz erwartete. Und das Ergebnis einer Methode, die eine Objektreferenz zurückgab, konnte auch nicht direkt einer primitiven Variablen zugewiesen werden – selbst wenn die zurückgegebene Referenz zu einem Integer gehörte und die primitive Variable ein int war. Es gab einfach keine Beziehung zwischen einem Integer und einem int, abgesehen davon, dass der Integer eine Instanzvariable des Typs int enthielt (für den primitiven Wert, den der Integer einpackt). Die ganze Arbeit blieb Ihnen überlassen.

## Eine ArrayList mit primitiven int-Werten

### *Ohne* Autoboxing (Java-Versionen *vor* 5.0)

```
public void hantierenMitZahlenAufAlteArt() {
  ArrayList zahlenListe = new ArrayList();
  zahlenListe.add(new Integer(3));
  Integer eins = (Integer) zahlenListe.get(0);
  int intEins = eins.intValue();
}
```

Erzeugen Sie eine ArrayList. (Erinnern Sie sich: Vor 5.0 konnten Sie den TYP nicht angeben, daher war eine ArrayList immer eine Object-Liste.)

Den primitiven Wert >>3<< können Sie nicht zur Liste hinzufügen, daher müssen Sie ihn erst mit einem Integer umwickeln.

Herausgegeben wird etwas vom Typ Object, aber man kann einen Cast auf Integer machen.

Zu guter Letzt können Sie den primitiven Wert aus dem Integer herausholen.

# Autoboxing verwischt die Grenze zwischen primitiven Datentypen und Objekten

Das in Java 5.0 neu hinzugekommene Autoboxing vollzieht die Umwandlung von einem primitiven Typ in ein Wrapper-Objekt *automatisch*!

Sehen wir mal, was passiert, wenn wir eine ArrayList mit int-Werten erzeugen wollen.

## Eine ArrayList mit primitiven int-Werten

### *Mit* Autoboxing (Java-Version 5.0 oder höher)

```
public void hantierenMitZahlenAufNeueArt() {
  ArrayList<Integer> zahlenListe = new ArrayList<Integer>();
  zahlenListe.add(3);
  int zahl = zahlenListe.get(0);
}
```

Erzeugen Sie eine ArrayList vom Typ Integer.

Einfach hinzufügen!

Es gibt zwar KEINE Methode für add(int) in ArrayList, aber der Compiler übernimmt die ganze Einpackerei (das Boxing) für Sie. Mit anderen Worten: In Wirklichkeit wird doch ein Integer-Objekt in der ArrayList gespeichert, aber Sie können »so tun«, als ob die ArrayList int-Werte annimmt. (Zu einer ArrayList<Integer> können Sie sowohl int- als auch Integer-Werte hinzufügen.)

Und der Compiler packt das Integer-Objekt auch automatisch aus (Unboxing), so dass Sie den int-Wert direkt einer primitiven Variablen zuweisen können, ohne die Methode intValue() auf dem Integer-Objekt aufrufen zu müssen.

**F: Warum kann man nicht eine ArrayList<int> deklarieren, wenn diese int-Werte enthalten soll?**

A: Weil ... *man es nicht kann.* Erinnern Sie sich – die Regel für generische Datentypen lautet, dass Sie nur Klassen- oder Interface-Typen angeben dürfen, *keine primitiven Typen.* ArrayList<int> lässt sich daher nicht kompilieren. Aber wie Sie an dem obigen Code sehen können, ist das eigentlich egal, denn der Compiler erlaubt es, dass Sie int-Werte in die ArrayList<Integer> stecken. Tatsächlich gibt es bei Verwendung eines Java 5.0-fähigen Compilers eigentlich nichts, was Sie daran *hindern* könnte, primitive Werte in eine ArrayList zu stecken, wenn deren Typ dem des entsprechenden Wrappers entspricht – das Autoboxing passiert nämlich automatisch. Sie können also primitive boolean-Werte in eine ArrayList<Boolean> stecken und char-Werte in eine ArrayList<Character>.

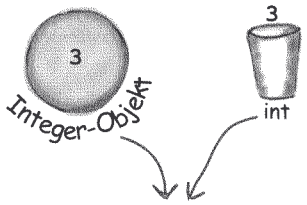
# Autoboxing funktioniert fast überall

Autoboxing eröffnet Ihnen noch mehr Möglichkeiten als das nahe liegende Ein- und Auspacken bei der Verwendung primitiver Werte in einer Collection ... Sie können damit auch einen primitiven Typ oder seinen Wrapper-Typ praktisch überall dort einsetzen, wo der jeweils andere erwartet wird. Denken Sie mal darüber nach!

## Spaß mit Autoboxing

### Methodenargumente

Wenn eine Methode einen Wrapper-Typ als Argument erwartet, können Sie ihr eine Referenz auf einen Wrapper oder auf einen primitiven Wert vom passenden Typ übergeben. Und natürlich gilt auch das Umgekehrte - wenn eine Methode einen primitiven Typ erwartet, können Sie ihr entweder einen passenden primitiven Wert oder eine Referenz auf einen Wrapper für diesen primitiven Typ übergeben.

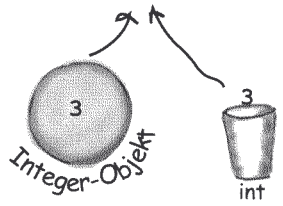



```
void nimmZahl(Integer i) {}
```

### Rückgabewerte

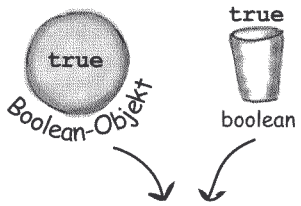
Wenn eine Methode einen primitiven Rückgabetyp deklariert, können Sie entweder einen passenden primitiven Wert oder eine Referenz auf einen Wrapper für diesen primitiven Typ zurückgeben. Und wenn eine Methode einen Wrapper-Rückgabetyp deklariert, können Sie entweder eine Referenz auf den Wrapper oder einen primitiven Wert des passenden Typs zurückgeben.

```
int gibZahl() {
  return x;
}
```




### Boolesche Ausdrücke

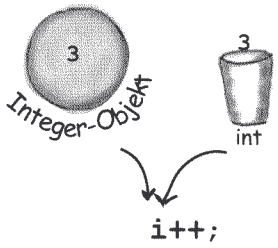
Überall dort, wo ein Boolescher Wert erwartet wird, können Sie entweder einen Ausdruck benutzen, der zu einem Booleschen Wert ausgewertet wird (4 > 2), oder einen primitiven boolean-Wert oder eine Referenz auf einen Boolean-Wrapper.




```
if (bool) {
  System.out.println("true");
}
```

## Operationen auf Zahlen

Dies ist wahrscheinlich das Merkwürdigste: Sie können jetzt tatsächlich einen Wrapper-Typ als Operanden in Operationen benutzen, in denen der primitive Typ erwartet wird. Sie können also beispielsweise den Inkrementoperator bei einer Referenz auf ein Integer-Objekt anwenden!




Aber seien Sie unbesorgt - das ist bloß ein Compiler-Trick. Die Sprache wurde nicht so modifiziert, dass die Operatoren mit Objekten arbeiten können, sondern der Compiler wandelt einfach vor der Operation das Objekt in seinen primitiven Typ um. Aber es sieht natürlich schon eigenartig aus.

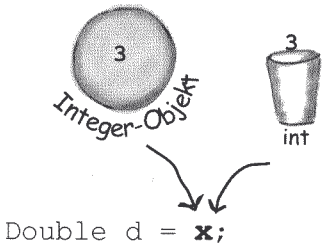
```
Integer i = new Integer(42);
i++;
```

Und das bedeutet, Sie können auch so etwas tun:

```
Integer j = new Integer(5);
Integer k = j + 3;
```

## Zuweisungen

Einer Variablen, die als Wrapper- oder als primitiver Typ deklariert ist, können Sie wahlweise einen passenden Wrapper oder einen primitiven Wert zuweisen. Beispielsweise kann eine primitive int-Variable einer Integer-Referenzvariablen zugewiesen werden und umgekehrt - eine Referenz auf ein Integer-Objekt kann einer Variablen zugewiesen werden, die als primitiver int-Typ deklariert wurde.




### Spitzen Sie Ihren Bleistift

Lässt sich dieser Code kompilieren? Und ausführen? Falls er läuft – was wird er machen?
Nehmen Sie sich Zeit, um darüber nachzudenken; es steckt ein mit dem Autoboxing verbundenes Problem dahinter, über das wir noch nicht gesprochen haben.
Sie müssen sich an Ihren Compiler wenden, um die Antworten zu bekommen. (Ja, wir zwingen Sie zum Experimentieren – nur zu Ihrem Besten, natürlich.)

```
public class TestBox{

  Integer i;
  int j;

  public static void main (String[] args) {
    TestBox t = new TestBox();
    t.los();
  }

  public void los() {
    j=i;
    System.out.println(j);
    System.out.println(i);
  }
}
```

# Warten Sie! Es kommt noch mehr! Wrapper besitzen auch statische Dienstmethoden!

Wrapper verhalten sich nicht nur wie normale Klassen, sondern haben auch noch eine Menge wirklich nützlicher statischer Methoden. Eine davon haben wir in diesem Buch schon benutzt – Integer.parseInt().

Die parse-Methoden nehmen einen String entgegen und geben Ihnen einen primitiven Wert zurück.

**Die Umwandlung eines Strings in einen primitiven Wert ist einfach:**

```
String s = "2";
int x = Integer.parseInt(s);
double d = Double.parseDouble("420.24");

boolean b = new Boolean("true").booleanValue();
```

Kein Problem, »2« wird durch Parsen zu 2 ausgewertet.

Sie hätten gedacht, es würde etwas wie Boolean.parseBoolean() geben, oder? Gibt es aber nicht. Glücklicherweise gibt es einen Boolean-Konstruktor, der einen String nimmt (und ihn parst). Den primitiven Wert erhalten Sie daraus dann einfach durch Auspacken.

**Aber wenn Sie das hier versuchen:**

```
String t = "two";
int y = Integer.parseInt(t);
```

Oje! Das lässt sich zwar wunderbar kompilieren, geht aber zur Laufzeit schief. Alles, was nicht durch Parsen in eine Zahl überführt werden kann, führt zu einer NumberFormatException.

**Sie erhalten eine Laufzeit-Exception:**

```
Datei Bearbeiten Fenster Hilfe Anhaltspunkt
% java Wrappers
Exception in thread "main"
java.lang.NumberFormatException: two
at java.lang.Integer.parseInt(Integer.java:409)
at java.lang.Integer.parseInt(Integer.java:458)
at Wrappers.main(Wrappers.java:9)
```

**Jede Methode und jeder Konstruktor kann beim Parsen eines Strings eine NumberFormatException auslösen. Es ist eine Laufzeit-Exception, daher müssen Sie sie nicht behandeln oder deklarieren. Aber manchmal möchten Sie das vielleicht.**

(Exceptions besprechen wir im nächsten Kapitel.)

# Und jetzt umgekehrt ... die Umwandlung einer primitiven Zahl in einen String

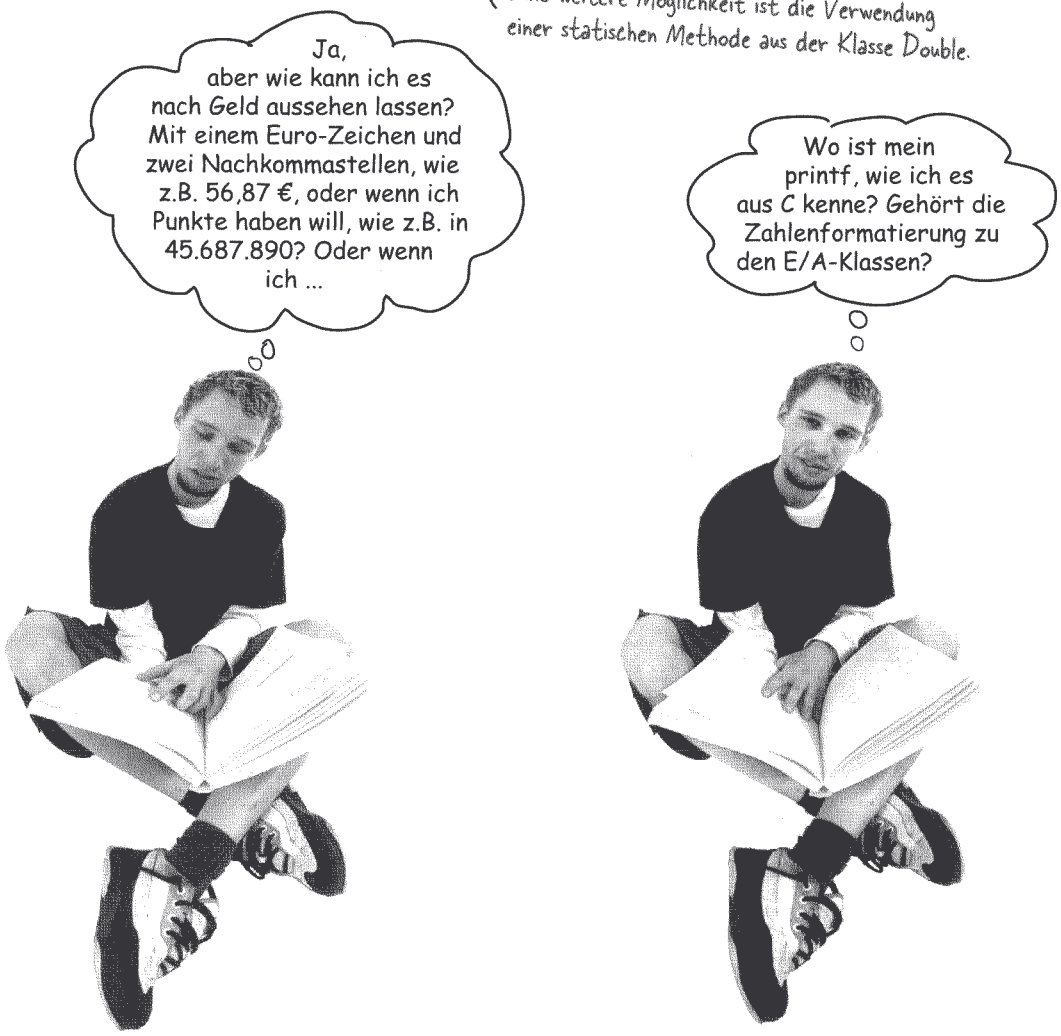
Es gibt verschiedene Möglichkeiten, eine Zahl in einen String zu verwandeln. Die einfachste ist die Verkettung der Zahl mit einem bestehenden String.

```
double d = 42.5;
String doubleString = "" + d;
```

Wie Sie wissen, ist der Operator + in Java überladen (als einziger Operator), so dass er auch als String-Verketter fungiert. Alles, was man an einen String anhängt, wird String-ifiziert.

```
double d = 42.5;
String doubleString = Double.toString(d);
```

Eine weitere Möglichkeit ist die Verwendung einer statischen Methode aus der Klasse Double.



# Formatierung von Zahlen

In Java muss die Formatierung von Zahlen und Datumswerten nicht an Eingabe/Ausgabe (E/A, engl. Input/Output, abgekürzt I/O) gekoppelt sein. Überlegen Sie: Eine der wichtigsten Möglichkeiten für die Anzeige von Zahlen ist eine GUI, eine grafische Benutzeroberfläche. Sie setzen Strings in einen scrollbaren Textbereich oder vielleicht in eine Tabelle. Wenn die Formatierung ausschließlich in print-Anweisungen eingebaut wäre, könnten Sie nie eine Zahl in einen hübschen String für die Anzeige auf einer Oberfläche umwandeln. Vor Java 5.0 wurde die Formatierung größtenteils durch Klassen im Paket java.text vorgenommen, aber die werden wir uns in dieser Auflage des Buchs noch nicht einmal mehr ansehen, weil die Dinge jetzt völlig anders liegen.

In Java 5.0 hat das Java-Team eine leistungsfähigere und flexiblere Formatierung durch die java.util-Klasse Formatter eingebaut. Allerdings müssen Sie selbst keine Methoden auf der Klasse Formatter aufrufen, weil Java 5.0 zu einigen der E/A-Klassen (unter anderem printf()) und zur Klasse String Methoden hinzugefügt hat, mit denen Sie es bequemer haben. Sie brauchen einfach nur eine statische Methode String.format() aufzurufen und ihr das Ding, das Sie formatiert haben möchten, zusammen mit den Formatierungsanweisungen zu übergeben.

Natürlich müssen Sie wissen, in welcher Form man die Formatierungsanweisungen liefert, und das erfordert ein bisschen Mühe, falls Sie nicht schon mit der Funktion ***printf()*** aus C/C++ vertraut sind. Doch selbst wenn Sie printf() *nicht* kennen, können Sie sich für die grundlegenden Dinge glücklicherweise nach einfachen Rezepten richten (die wir Ihnen in diesem Kapitel zeigen). Wenn Sie jedoch wirklich *alles* genau wie gewünscht hinbekommen möchten, sollten Sie das Formatieren von Grund auf lernen.

Wir beginnen hier mit einem einfachen Beispiel und sehen uns dann an, wie es funktioniert.

## Die Formatierung einer Zahl mit Trennpunkten

```
public class TestFormatierung {

  public static void main(String[] args){

    String s = String.format("%, d", 1000000000);
    System.out.println(s);
  }
}
```

Die zu formatierende Zahl (wir wollen, dass sie Tausendertrennpunkte erhält).

Die Anweisungen dazu, wie das zweite Argument (in diesem Fall ein int-Wert) zu formatieren ist. Denken Sie daran, die Methode erhält hier nur zwei Argumente – das erste Komma steht nämlich INNERHALB des String-Literals, trennt also keine Argumente der Formatierungsmethode.

```
1.000.000.000
```

Jetzt wird die Zahl durch Punkte gegliedert ausgegeben.

# Formatierung in ihre Einzelteile zerlegt ...

Grundsätzlich besteht die Formatierung aus zwei Hauptbestandteilen (es gibt noch mehr, aber wir wollen es für den Anfang möglichst einfach halten):

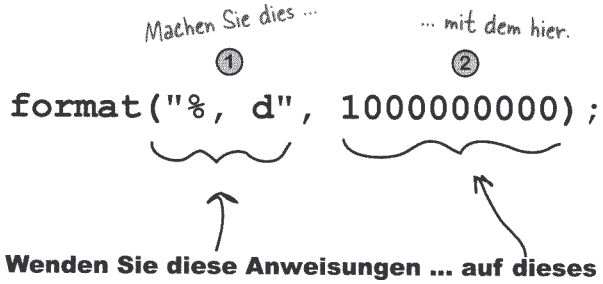
**1 Formatierungsanweisungen**

Sie verwenden spezielle Formatspezifizierer, die beschreiben, wie das Argument formatiert werden soll.

**2 Das zu formatierende Argument**

Es kann zwar mehr als ein Argument geben, aber wir fangen mit nur einem an. Der Argumenttyp kann nicht einfach *irgendetwas* sein ... es muss etwas sein, was sich mit den Formatspezifizierern in den Formatierungsanweisungen formatieren lässt. Wenn sich Ihre Formatierungsanweisungen z.B. auf eine *Gleitkommazahl* beziehen, können Sie keinen Hund übergeben und noch nicht einmal einen String, der wie eine Gleitkommazahl aussieht.

Anmerkung: Wenn Sie printf() aus C/C++ kennen, können Sie die nächsten paar Seiten wahrscheinlich einfach überfliegen. Wenn nicht, lesen Sie alles sorgfältig!




### Was sagen diese Anweisungen eigentlich?

»Nimm das zweite Argument dieser Methode, formatiere es als dezimale Ganzzahl und füge **Tausendertrennzeichen** ein.«

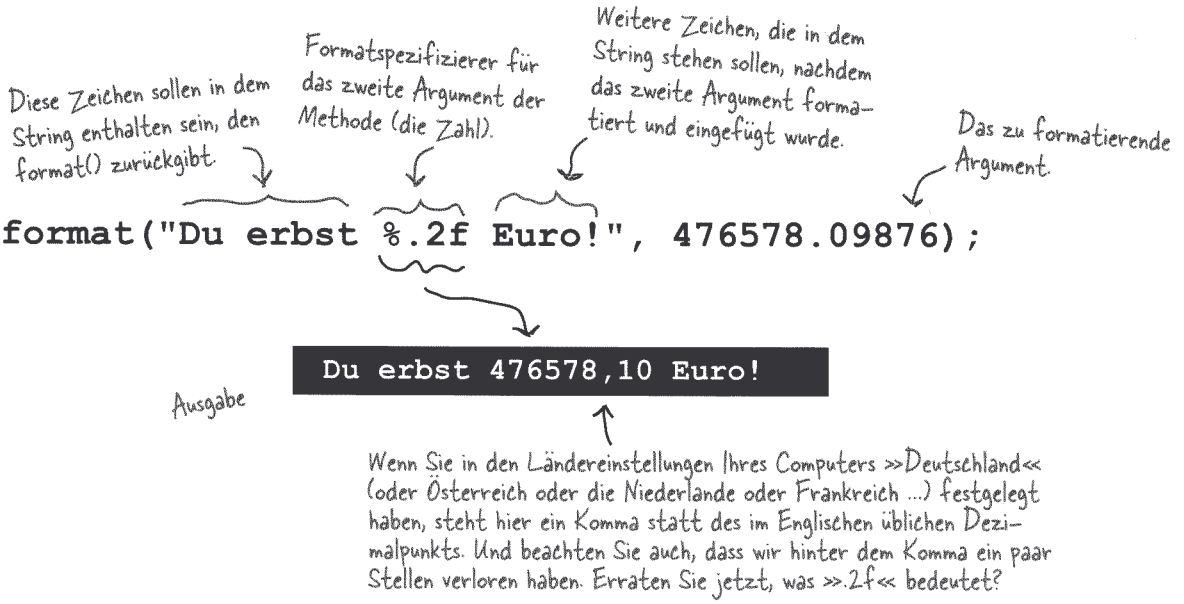
Sie fragen sich, warum in den Anweisungen ein Komma steht und kein Punkt? Ganz einfach: Im englischen Sprachraum benutzt man das Komma als Tausendertrennzeichen. Im deutschen Sprachraum verwendet man stattdessen Punkte (vor allem bei Geldbeträgen) oder Leerzeichen. Und wenn Sie in den lokalen Einstellungen Ihres Computers z.B. »Deutschland« oder »Österreich« festgelegt haben, gibt Ihnen die Methode format() automatisch Zahlen zurück, die durch Punkte getrennt sind. Aber probieren Sie doch einfach einmal aus, was bei anderen Ländereinstellungen passiert ...

### Und wie sagen sie das?

Auf der nächsten Seite sehen wir uns noch genauer an, was die Syntax »%, d« genau bedeutet, aber so viel schon einmal vorweg: Immer wenn Sie in einem Format-String (dem ersten Argument der Methode format()) ein Prozentzeichen (%) entdecken, stellen Sie sich vor, dass es eine Variable repräsentiert und dass das andere Argument der Methode diese Variable ist. Die übrigen Zeichen hinter dem Prozentzeichen beschreiben die Formatierungsanweisungen für das Argument.

# Das %-Zeichen bedeutet: »hier Argument einfügen«
## (und es mit diesen Anweisungen formatieren)

Das erste Argument der Methode format() wird als Format-String bezeichnet, und dieser kann tatsächlich Zeichen enthalten, die Sie einfach so ausgeben möchten, wie sie sind, ohne sie extra zu formatieren. Wenn Sie aber darin das %-Zeichen entdecken, sehen Sie es als eine Variable an, die das andere Argument der Methode repräsentiert.




Das %-Zeichen sagt dem Formatierer, dass hier das andere Argument dieser Methode (das zweite Argument von format(), also die Zahl) eingefügt werden soll UND dass die Formatierung mit den Zeichen ».2f« hinter dem Prozentzeichen erfolgen soll. Dann wird für die Ausgabe noch der Rest des Format-Strings (» Euro!«) angehängt.

### Ein Trennzeichen einfügen

```
format("Du erbst %,.2f Euro!", 476578.09876);
```

```
Du erbst 476.578,10 Euro!
```

Wir haben die Formatanweisung von >>%.2f<< in >>%,.2f<< geändert, also ein Komma eingefügt, weil wir ein Tausendertrennzeichen haben wollten. Deshalb steht jetzt ein Trennpunkt in der formatierten Zahl.

## Der Format-String verwendet seine eigene kleine Sprachsyntax

Ganz offensichtlich können Sie nicht einfach etwas *Beliebiges* hinter das %-Zeichen schreiben. Die Syntax für das, was hinter dem Prozentzeichen steht, folgt ganz speziellen Regeln und beschreibt, wie das Argument zu formatieren ist, das an dieser Stelle in den (formatierten) Ergebnis-String eingefügt wird.

Zwei Beispiele haben Sie schon gesehen:

**%, d** heißt: »Füge Tausendertrennzeichen ein und formatiere die Zahl als dezimale Ganzzahl.«

und

**%.2f** heißt: »Formatiere die Zahl als Gleitkommazahl mit einer Genauigkeit von zwei Nachkommastellen.«

und

**%,.2f** heißt: »Füge Tausendertrennzeichen ein und formatiere die Zahl als Gleitkommazahl mit einer Genauigkeit von zwei Nachkommastellen.«

Die eigentliche Frage lautet aber: »Woher weiß ich, was ich hinter das Prozentzeichen setzen muss, damit der Formatierer das tut, was ich will?« Und dazu gehört auch die Kenntnis der Symbole (wie »d« für dezimal und »f« für Gleitkomma) und der Reihenfolge, in der die Anweisungen hinter dem Prozentzeichen zu platzieren sind. Wenn Sie z.B. das Komma hinter das »d« setzen, also »%d,« statt »%,d« schreiben, funktioniert es nicht!

Oder doch? Was macht Ihrer Ansicht nach das hier:

```
String.format("Du erbst %.2f, Euro!", 476578.09876);
```

(Die Antwort geben wir auf der nächsten Seite.)

# Der Formatspezifizierer

Alles, was hinter dem Prozentzeichen steht, bis einschließlich des Datentyp-Anzeigers (wie »d« oder »f«), gehört zu den Formatierungsanweisungen. Für die auf den Typanzeiger folgenden Zeichen nimmt der Formatierer an, dass sie Teil des Ausgabe-Strings sein sollen – solange er nicht auf ein weiteres Prozentzeichen trifft. Hmmm ... wäre das denn überhaupt möglich? Kann es mehr als eine formatierte Argumentvariable geben? Schieben Sie diesen Gedanken vorläufig beiseite; wir kommen in ein paar Minuten darauf zurück. Jetzt sehen wir uns erst einmal die Syntax für die Formatspezifizierer an – die Dinger hinter dem Prozentzeichen – und beschreiben, wie das Argument formatiert werden sollte.

**Ein Formatspezifizierer kann aus bis zu fünf verschiedenen Teilen bestehen (das »%« nicht mitgezählt). Alles, was im Folgenden in eckigen Klammern steht, ist optional, d.h., nur das Prozentzeichen (%) und der Typ sind unbedingt erforderlich. Doch auch die Reihenfolge ist vorgeschrieben, d.h., alle Teile, die Sie verwenden, müssen in dieser Reihenfolge stehen.**

```
%[Argumentnummer][Schalter][Breite][.Genauigkeit]Typ
```

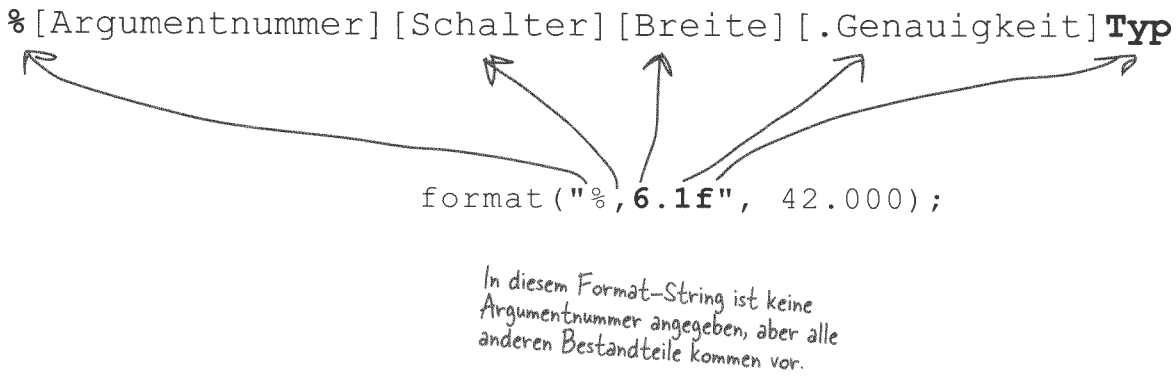
Dazu kommen wir später ... Sie können damit angeben, WELCHES Argument Sie meinen, falls es mehr als eines gibt. (Machen Sie sich darüber jetzt noch keine Gedanken.)

Diese Schalter oder »Flags« signalisieren dem Formatspezifizierer, dass er spezielle Formatierungsoptionen berücksichtigen soll, wie das Einfügen von Tausendertrennzeichen, das Klammern von negativen Zahlen oder die linksbündige Ausrichtung von Zahlen.

Dies definiert die MINDESTanzahl der Zeichen, die verwendet werden. Wohlgemerkt: die Mindest-, nicht die Gesamtanzahl. Ist die Zahl länger als die angegebene Breite, wird sie vollständig ausgegeben, ist sie jedoch kürzer, wird sie mit Nullen aufgefüllt.

Das kennen Sie schon ... es legt die Genauigkeit fest, also die Anzahl der Nachkommastellen. Vergessen Sie nicht, den ».« davorzusetzen.

Die Angabe des Typs ist zwingend erforderlich (siehe nächste Seite). Normalerweise ist es »d« für eine dezimale Ganzzahl oder »f« für eine Gleitkommazahl.




In diesem Format-String ist keine Argumentnummer angegeben, aber alle anderen Bestandteile kommen vor.

# Zwingend erforderlich ist nur der TYP-Spezifizierer

Der Typ ist der einzige Spezifizierer, den Sie angeben *müssen*, aber vergessen Sie nicht: Wenn Sie daneben noch etwas anderes angeben, muss der Typ immer als Letztes stehen. Es gibt mehr als ein Dutzend verschiedener Typmodifizierer (ohne Berücksichtigung von Datums- und Zeitangaben, die ihre eigenen Modifizierer haben). Meistens werden Sie aber wohl %d (Dezimalzahl) oder %f (Gleitkommazahl) verwenden. Und %f kombinieren Sie typischerweise mit einer Genauigkeitsangabe, mit der Sie die Anzahl der Nachkommastellen für Ihre Ausgabe festlegen.

**Der TYP muss angegeben werden, alles andere ist freigestellt.**

**%d** ***Dezimalzahl***

```
format("%d", 42);
```

```
42
```

Mit 42.25 würde es nicht funktionieren! Das wäre das Gleiche wie der Versuch, einen double-Typ direkt einer int-Variablen zuzuweisen.

Das Argument muss mit einem int kompatibel sein, es gehen also nur byte, short, int und char (oder ihre Wrapper-Typen).

---

**%f** ***Gleitkommazahl***

```
format("%.3f", 42.000000);
```

```
42,000
```

Hier haben wir das »f« mit der Genauigkeitsangabe ».3« kombiniert und daher drei Nullen erhalten.

Das Argument muss ein Gleitkommatyp sein, daher kommen nur float oder double in Frage (als primitive Typen oder Wrapper) oder etwas, das BigDecimal heißt (und das wir uns in diesem Buch nicht ansehen).

---

**%x** ***Hexadezimalzahl***

```
format("%x", 42);
```

```
2a
```

Als Argument kommen byte, short, int, long (primitive oder Wrapper-Typen) oder BigInteger in Frage.

---

**%c** ***Zeichen***

```
format("%c", 42);
```

```
*
```

Die Zahl 42 repräsentiert das Zeichen »*«.

Als Argument kommen byte, short, char oder int (primitive oder Wrapper-Typen) in Frage.

Sie müssen in Ihren Formatanweisungen einen Typ angeben; wenn Sie daneben noch anderes angeben, muss der Typ immer als Letztes stehen. Meistens werden Sie Zahlen aber wohl nur mit »d« für Dezimalzahlen oder »f« für Gleitkommazahlen formatieren.

# Was passiert, wenn ich mehr als ein Argument habe?

Stellen Sie sich vor, Sie wollen einen String, der so aussieht:

»Der Betrag steigt von *20.456.654* auf *100.567.890,25*.«

Aber – die Zahlen stammen aus Variablen. Also, was machen Sie? Sie geben hinter dem Format-String (dem ersten Argument) noch *zwei* weitere Argumente an, übergeben beim Aufruf von format() also drei Argumente statt nur zwei. Und innerhalb des ersten Arguments (des Format-Strings) haben Sie zwei verschiedene Formatspezifizierer (zwei von den Dingern, die mit »%« anfangen). Der erste Formatspezifizierer fügt dann das zweite Argument der Methode ein und der zweite Formatspezifizierer das dritte Argument der Methode. Anders ausgedrückt: Das Einfügen der Variablen in den Format-String erfolgt in der Reihenfolge, in der die anderen Argumente der Methode format() übergeben werden.

```
int eins = 20456654;
double zwei = 100567890.248907;
String s = String.format("Der Betrag steigt von %,d auf %,.2f", eins, zwei");
```

```
Der Betrag steigt von 20.456.654 auf 100.567.890,25.
```

Wenn Sie mehrere Argumente haben, werden diese in der Reihenfolge eingefügt, in der sie der Methode format() übergeben werden.

Wir haben in beide Variablen Trennpunkte eingefügt und die Gleitkommazahl (die zweite Variable) auf zwei Stellen hinterm Komma begrenzt.

Wenn wir zur Datumsformatierung kommen, werden Sie sehen, dass es Situationen geben kann, in denen Sie verschiedene Formatspezifizierer auf das gleiche Argument anwenden möchten. Wahrscheinlich können Sie sich das nur schwer vorstellen, solange Sie nicht gesehen haben, wie *Datums*formatierung (im Gegensatz zur bisher besprochenen *Zahlen*formatierung) funktioniert. Wir werden Ihnen aber gleich noch zeigen, wie man genauer angeben kann, welche Formatspezifizierer auf welche Argumente angewendet werden.

**F: Ähm, hier passiert ja was ECHT Seltsames ... Wie viele Argumente *kann* ich denn eigentlich übergeben? Ich meine, wie viele überladene format()-Methoden gibt es denn IN der Klasse String? Was passiert, wenn ich, sagen wir mal, zehn verschiedene Argumente übergeben will, die für einen einzigen Ausgabe-String formatiert werden sollen?**

A: Gut erkannt! Ja, da passiert wirklich etwas Seltsames (oder zumindest etwas ganz Neues). Und: Nein, es gibt *nicht* haufenweise überladene format()-Methoden, die eine unterschiedliche Anzahl von möglichen Argumenten entgegennehmen. Damit diese neue (printf-ähnliche) Formatierungs-API in Java unterstützt wird, brauchte die Sprache eine weitere neue Eigenschaft – *variable Argumentlisten* (kurz *Varargs* genannt). Varargs besprechen wir nicht ausführlich, denn in einem System mit einem guten Design werden Sie diese – außer für Formatierungszwecke – vermutlich nicht benutzen.

# So viel zu Zahlen – und was ist mit Datumsangaben?

Stellen Sie sich vor, Ihr String soll so aussehen: »Mittwoch, 21. Dez. 2005«.

Nichts Besonderes, sagen Sie? Na, nun stellen Sie sich mal vor, Ihnen steht dafür nichts weiter zur Verfügung als eine Variable vom Typ Date – eine Java-Klasse, die einen Zeitstempel repräsentieren kann –, und jetzt wollen Sie dieses Objekt (an Stelle einer Zahl) durch den Formatierer schicken.

Der Hauptunterschied zwischen Zahlen- und Datumsformatierung besteht darin, dass Datumsformate einen aus zwei Zeichen bestehenden Typ verwenden, der mit »t« beginnt (statt des Einzelbuchstabens wie z.B. »f« bzw. »d«). Die folgenden Beispiele vermitteln ein ganz gutes Bild davon, wie es funktioniert:

### Vollständige Datums- und Zeitangabe %tc

```
String.format("%tc", new Date());
```

```
Mi Dez 21 16:41:39 CET 2005
```

---

### Nur die Zeit %tr

```
String.format("%tr", new Date());
```

```
04:41:51 PM
```

---

### Wochentag, Tag und Monat %tA %td %tB

Es gibt keinen Formatspezifizierer, der das, was wir haben wollen, allein hinbekommt. Wir müssen daher drei Spezifizierer kombinieren, nämlich für den Wochentag (%tA), den Tag im Monat (%td) und den Monat (%tB).

```
Date heute = new Date();
String.format("%tA, %td. %tB",heute,heute,heute);
```

Das Komma gehört nicht zur Formatierung – das ist nur das Zeichen, das wir hinter dem ersten formatierten Argument ausgegeben haben wollen. Das Gleiche gilt für den Punkt hinter »%td«.

Das heißt aber, wir müssen das Date-Objekt dreimal übergeben, je einmal für jeden von uns gewünschten Formatierungsteil. Anders ausgedrückt: Mit %tA kriegen wir nur den Wochentag, aber dann brauchen wir es noch mal, nur um den Tag im Monat zu bekommen, und dann noch mal für den Monat.

```
Mittwoch, 21. Dezember
```

---

### Das Gleiche noch mal, aber *ohne Mehrfachangabe des Arguments* %tA %td %tB

```
Date heute = new Date();
String.format("%tA, %<td. %<tB",heute);
```

Sie können sich das so vorstellen, als ob drei verschiedene Getter-Methoden auf das Datumsobjekt zugreifen, um drei verschiedene Datenbröckchen herauszuholen.

Die spitze Klammer < ist einfach ein weiterer Schalter im Spezifizierer, der dem Formatierer sagt: »Nimm das letzte Argument noch mal.« Das erspart Ihnen also die Wiederholung der Argumente; stattdessen formatieren Sie das gleiche Argument auf drei unterschiedliche Arten.

## Arbeiten mit Datumswerten

Mit Datumswerten muss man noch mehr machen können, als nur an das Datum von *heute* zu gelangen. Ihre Programme müssen Datumswerte anpassen, verstrichene Zeitspannen ermitteln, Prioritäten in Plänen setzen können – ach was, sie müssen Pläne erstellen können! Industrietaugliche Datumsmanipulationsfähigkeiten brauchen Sie.

Nun könnten Sie natürlich Ihre eigenen Datumsroutinen erstellen ... (und vergessen Sie nicht die Schaltjahre!). Und – auweia! – diese ärgerlichen gelegentlichen Schalt*sekunden*. Wahnsinn, das könnte kompliziert werden. Aber wir haben hier eine gute Neuigkeit: Die Java-API bietet eine große Auswahl an Klassen, die Ihnen beim Umgang mit Datumswerten behilflich sein können. Manchmal ja fast *ein bisschen viel* Auswahl ...

# In der Zeit vor- und zurückspringen

Nehmen wir an, der Arbeitsplan in Ihrer Firma geht von Montag bis Freitag. Sie haben die Aufgabe, den letzten Arbeitstag für jeden Monat in diesem Jahr herauszufinden ...

### Anscheinend ist java.util.Date irgendwie ... nicht up to date

Weiter oben haben wir java.util.Date benutzt, um an das Datum von heute zu kommen. Es liegt daher nahe, sich erst einmal bei dieser Klasse nach praktischen Möglichkeiten zur Datumsmanipulation umzusehen. Aber wenn Sie in der API nachschlagen, stellen Sie fest, dass die meisten Methoden von Date als »deprecated« gekennzeichnet sind, d.h., dass von ihrer Verwendung abgeraten wird!

Die Klasse Date eignet sich trotzdem prima, wenn man einen »Zeitstempel« braucht – ein Objekt, das das aktuelle Datum und die aktuelle Zeit repräsentiert. Benutzen Sie sie daher, wenn Sie sagen wollen: »Gib mir ein JETZT!«

Die Java-API empfiehlt, stattdessen **java.util.Calendar** zu verwenden; sehen wir uns das also mal an:

Wenn Sie einen Zeitstempel für »jetzt« brauchen, nehmen Sie Date. Aber für alles andere verwenden Sie Calendar.

### Benutzen Sie java.util.Calendar für Ihre Datums*manipulationen*

Die Designer der Calendar-API wollten im wörtlichen Sinn global denken. Die Grundidee ist, dass Sie für das Arbeiten mit Datumsangaben um einen »Kalender« bitten (über eine statische Methode der Klasse Calendar, die Sie auf der nächsten Seite kennen lernen) und die JVM Ihnen dann eine Instanz einer konkreten Unterklasse von Calendar aushändigt. (Calendar ist eine abstrakte Klasse, daher arbeiten Sie immer mit einer konkreten Unterklasse.)

Noch interessanter ist jedoch, dass die *Art* des Kalenders, die Sie erhalten, *zu Ihren lokalen Einstellungen passt.* Ein großer Teil der Welt benutzt den gregorianischen Kalender, aber für Gegenden, in denen kein gregorianischer Kalender benutzt wird, gibt es Java-Bibliotheken, die mit anderen Kalendern arbeiten, z.B. mit dem buddhistischen oder islamischen oder japanischen.

Die Java-Standard-API wird mit **java.util.GregorianCalendar** ausgeliefert, deshalb benutzen wir hier diesen. Meistens brauchen Sie jedoch keinen Gedanken daran zu verschwenden, welche Art von Calendar-Unterklasse Sie benutzen, und können sich stattdessen nur auf die Methoden der Klasse Calendar konzentrieren.

# Wie man an ein Objekt kommt, das Calendar erweitert

Wie um alles in der Welt kommen Sie denn nun an eine »Instanz« einer abstrakten Klasse?

Gar nicht, natürlich – das hier geht ja nicht:

**Das geht NICHT:**

```
Calendar cal = new Calendar();
```

Das erlaubt der Compiler nicht!

**Verwenden Sie stattdessen die statische Methode »getInstance()«:**

```
Calendar cal = Calendar.getInstance();
```

Diese Syntax sollte Ihnen mittlerweile vertraut vorkommen – wir rufen eine statische Methode auf.

## Sie können keine Instanz von Calendar bekommen, aber eine Instanz einer konkreten Unterklasse von Calendar.

Offensichtlich können Sie keine Instanz von Calendar bekommen, weil Calendar abstrakt ist. Aber Sie dürfen immerhin noch statische Methoden auf Calendar aufrufen, da *statische* Methoden auf der *Klasse* aufgerufen werden statt auf einer bestimmten Instanz. Also rufen Sie die statische Methode getInstance() auf Calendar auf, und diese gibt Ihnen ... eine Instanz einer konkreten Unterklasse zurück. Irgendetwas, das Calendar erweitert (was bedeutet, dass es Calendar polymorph zugewiesen werden kann) und das – vertragsgemäß – auf einen Aufruf der Methoden der Klasse Calendar reagieren kann.

Im größten Teil der Welt und standardmäßig in den meisten Java-Versionen erhalten Sie eine Instanz von **java.util.GregorianCalendar.**

# Arbeiten mit Calendar-Objekten

Um mit Calendar-Objekten arbeiten zu können, müssen Sie ein paar zu Grunde liegende Konzepte verstehen:

- **Felder halten *den Zustand* fest.** Ein Calendar-Objekt hat viele Felder, die Aspekte seines gegenwärtigen Zustands – des Datums und der Zeit – repräsentieren. Beispielsweise können Sie über Getter und Setter an *Jahr* oder *Monat* eines Kalenders herankommen.
- **Datums- und Zeitwerte können *inkrementiert* werden.** Die Klasse Calendar enthält Methoden, mit denen Sie in verschiedenen Feldern Werte addieren oder subtrahieren können, z.B. »erhöhe den Monat um 1« oder »ziehe 3 Jahre ab«.
- **Datums- und Zeitangaben können in *Millisekunden* dargestellt werden.** Mit der Klasse Calendar können Sie Ihre Datumswerte in eine Millisekunden-Repräsentation konvertieren und wieder zurück. (Genauer gesagt, in die Anzahl der Millisekunden, die seit dem 1. Januar 1970 vergangen sind.) Damit können Sie genaue Berechnungen durchführen, z.B. »ermittle die zwischen zwei Zeitpunkten verstrichene Zeit« oder »addiere 63 Stunden, 23 Minuten und 12 Sekunden zu diesem Zeitpunkt hinzu«.

### Ein Beispiel für das Arbeiten mit einem Calendar-Objekt:

```
Calendar c = Calendar.getInstance();
c.set(2004,0,7,15,40);
long tag1 = c.getTimeInMillis();
tag1 += 1000 * 60 * 60;
c.setTimeInMillis(tag1);
System.out.println("neue Stunde: " + c.get(c.HOUR_OF_DAY));
c.add(c.DATE, 35);
System.out.println("35 Tage hinzurechnen: " + c.getTime());
c.roll(c.DATE, 35);
System.out.println("35 Tage rollieren: " + c.getTime());
c.set(c.DATE, 1);
System.out.println("auf 1 setzen: " + c.getTime());
```

Setzen Sie die Zeit auf den 7. Januar 2004, 15 Uhr 40. (Beachten Sie die bei null beginnende Monatszählung.)

Wandeln Sie das in einen Riesenhaufen von Millisekunden um.

Addieren Sie die Millisekundenanzahl einer Stunde und aktualisieren Sie damit die Zeit. (Zur Erinnerung: »+=« bedeutet dasselbe wie tag1 = tag1 + ...)

Addieren Sie 35 Tage zum Datum, dadurch werden wir in den Februar verschoben.

»Rollieren« Sie von diesem Datum an 35 Tage. Das Datum wird dadurch 35 Tage vorgedreht, aber der Monat BLEIBT UNVERÄNDERT!

Hier wird nicht inkrementiert, sondern das Datum einfach »gesetzt«.

```
Datei Bearbeiten Fenster Hilfe DieZeitVerfliegt
neue Stunde: 16
35 Tage hinzurechnen: Wed Feb 11 16:40:07 CET 2004
35 Tage rollieren: Tue Feb 17 16:40:07 CET 2004
auf 1 setzen: Sun Feb 01 16:40:07 CET 2004
```

Diese Ausgabe zeigt, wie millis, add, roll und set funktionieren.

# Die Highlights der Calendar-API

Wir haben eben schon einige Felder und Methoden der Klasse Calendar durchgearbeitet. Da das eine große API ist, zeigen wir Ihnen nur die gebräuchlichsten Felder und Methoden, die Sie benutzen werden. Wenn Sie erst einmal ein paar davon kennen, wird es Ihnen leicht fallen, sich auch den Rest der API zu erobern.

## Die wichtigsten Calendar-Methoden

**add(int field, int amount)**
Addiert oder subtrahiert Zeit zum/vom Feld des Kalenders.

**get(int field)**
Gibt den Wert des betreffenden Kalenderfelds zurück.

**getInstance()**
Gibt einen Calendar zurück; Sie können ein Gebietsschema angeben (z.B. als Kombination aus Sprache und Land).

**getTimeInMillis()**
Gibt die Zeit dieses Kalenders in Millisekunden als long-Wert zurück.

**roll(int field, boolean up)**
Addiert oder subtrahiert Zeit, ohne größere Felder zu ändern.

**set(int field, int value)**
Setzt den Wert des betreffenden Calendar-Felds.

**set(year, month, day, hour, minute) (alle ints)**
Eine häufige Variante von set, um die Zeit völlig neu zu setzen.

**setTimeInMillis(long millis)**
Setzt die Zeit eines Kalenders mit einem Millisekunden-long-Wert.

**// mehr ...**

## Die wichtigsten Calendar-Felder

**DATE / DAY_OF_MONTH**
Tag innerhalb des Monats abfragen oder setzen.

**HOUR / HOUR_OF_DAY**
12-Stunden-Wert (mit Zusatz »AM« oder »PM«) bzw. 24-Stunden-Wert abfragen oder setzen.

**MILLISECOND**
Millisekunden abfragen oder setzen.

**MINUTE**
Minute abfragen oder setzen.

**MONTH**
Monat abfragen oder setzen.

**YEAR**
Jahr abfragen oder setzen.

**ZONE_OFFSET**
Zeit-Offset (ohne Sommerzeitberücksichtigung) gegenüber GMT in Millisekunden abfragen oder setzen.

**// mehr ...**

# Und noch mehr Statisches: statische Importe!

Neu in Java 5.0 ... und ein zweischneidiges Schwert. Manche mögen diese Idee, andere gar nicht. Der einzige Zweck von statischen Importen ist es, Ihnen Tipparbeit zu ersparen. Wenn Sie nicht gern tippen, gefällt Ihnen diese Möglichkeit vielleicht. Die Kehrseite statischer Importe ist, dass Ihr Quellcode damit – wenn Sie nicht aufpassen – viel schwerer lesbar wird.

Die Grundidee ist: Wann immer Sie eine statische Klasse, eine statische Variable oder eine Enumeration (mehr darüber später) verwenden, können Sie diese importieren und sich Tipparbeit ersparen.

**Sparsam einsetzen:**

**Statische Importe können Ihren Code schwer lesbar machen.**

**Code nach guter alter Art:**

```
import java.lang.Math;

class OhneStatic {

  public static void main(String [] args) {

    System.out.println("Wurzel: " + Math.sqrt(2.0));

    System.out.println("tan: " + Math.tan(60));

  }

}
```

**Der gleiche Code mit statischen Importen:**

Die Syntax für die Deklaration statischer Importe.

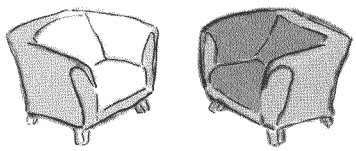
```
import static java.lang.System.out;

import static.java.lang.Math.*;

class MitStatic {

  public static void main(String [] args) {

    out.println("Wurzel: " + sqrt(2.0));

    out.println("tan: " + tan(60));

  }

}
```

Statische Importe in Aktion.

**Vorbehalte & Fallstricke**

- Wenn Sie ein statisches Klassenelement (Member) nur wenige Male verwenden wollen, sollten Sie unserer Ansicht nach statische Importe vermeiden, damit der Code besser lesbar bleibt.
- Wenn Sie ein statisches Member oft brauchen werden (also z.B. viele Math-Berechungen durchführen wollen), ist der statische Import wahrscheinlich sinnvoll.
- Beachten Sie, dass Sie in der Deklaration des statischen Imports auch Platzhalter (.*) verwenden können.
- Ein großes Problem bei statischen Importen ist, dass man ziemlich leicht Namenskonflikte zwischen Bezeichnern heraufbeschwört. Wenn Sie beispielsweise zwei verschiedene Klassen mit einer Methode »add()« haben – woher sollen Sie und der Compiler wissen, welche verwendet werden soll?

## Kamingespräche



Heute Abend: **Eine Instanzvariable hackt auf einer statischen Variablen herum**

**Instanzvariable**

Ich weiß wirklich nicht, was das hier soll. Jeder weiß doch, dass statische Variablen nur für Konstanten benutzt werden. Und wie viele davon gibt es? Ich schätze, in der ganzen API vielleicht vier? Und als ob die jemals irgendeiner verwendet.

Voll. Ja, das kannst du gern laut sagen. Okay, es gibt also ein paar in der Swing-Bibliothek, aber jeder weiß doch, dass Swing nur ein Sonderfall ist.

Okay, aber mal abgesehen von ein paar GUI-Sachen – nenn mir nur eine einzige statische Variable, die irgendjemand tatsächlich benutzen würde. Im wirklichen Leben.

Na ja, auch nur ein Sonderfall. Und das benutzt ja auch keiner, außer fürs Debugging.

**Statische Variable**

Du solltest wirklich erst mal die Fakten überprüfen. Wann hast du dir das letzte Mal die API angesehen? Sie ist nämlich verdammt voll mit statischen Sachen! Da gibt es sogar ganze Klassen, die nur dafür da sind, konstante Werte zu enthalten. Zum Beispiel eine Klasse namens SwingConstants, die ist randvoll damit.

Das mag ja ein Sonderfall sein, aber ein wirklich wichtiger! Und was ist mit der Klasse Color? Wäre das nicht ätzend, wenn man sich die RGB-Werte für die Standardfarben merken müsste? Aber in der Klasse Color sind schon Konstanten für Blau, Violett, Weiß, Rot usw. vordefiniert. Sehr praktisch.

Wie wäre es da gleich als Erstes mit System.out? Das out in System.out ist eine statische Variable der Klasse System. Du erzeugst nicht selbst eine neue Instanz des Systems, sondern fragst einfach die Klasse System nach ihrer out-Variablen.

Ach, als ob Debugging nicht wichtig wäre!

Und hier ist noch etwas, das dir mit deinem beschränkten Horizont vermutlich noch nie in den Sinn gekommen ist – seien wir mal ehrlich, statische Variablen sind effizienter. Eine pro Klasse statt eine pro Instanz. So könnte man Unmengen an Speicherplatz sparen!

**Instanzvariable**

Ähm, vergisst du da nicht was?

Statische Variablen sind doch so un-OO wie nur was!! Meine Güte, warum machen wir nicht einfach einen Riesenschritt rückwärts und programmieren ein bisschen prozedural, wenn wir schon mal dabei sind.

Du bist wie eine globale Variable, und jeder Programmierer, der seinen PDA wert ist, weiß doch, dass das normalerweise was ganz Schlimmes ist.

Ja, du lebst in einer Klasse, aber es heißt ja nicht *Klassen*-orientierte Programmierung. Das wäre auch blöd. Du bist ein Relikt. Etwas, das den alten Hasen helfen soll, auf Java umzusteigen.

Na gut, ab und zu ist es sicher sinnvoll, etwas Statisches zu benutzen, aber lass dir gesagt sein, der Missbrauch von statischen Variablen (und Methoden) ist ein Zeichen für einen unreifen OO-Programmierer. Ein Designer sollte über den *Objekt*zustand nachdenken, nicht über den *Klassen*zustand.

Statische Methoden sind das Schlimmste überhaupt, denn das heißt normalerweise, dass der Programmierer prozedural denkt und nicht daran, wie Objekte etwas auf der Basis ihres unverwechselbaren Objektzustands tun.

Ist ja guuuut. Glaub', was Du willst …

**Statische Variable**

Was denn?

Was meinst du mit *un*-OO?

Ich BIN KEINE globale Variable. So etwas gibt es nicht. Ich lebe in einer Klasse! Das ist ganz schön OO, weißt du, eine KLASSE. Ich sitze nicht irgendwo da draußen im Weltall; ich bin ein natürlicher Bestandteil vom Zustand eines Objekts; der einzige Unterschied ist, dass ich von allen Instanzen einer Klasse gemeinsam benutzt werde. Sehr effizient.

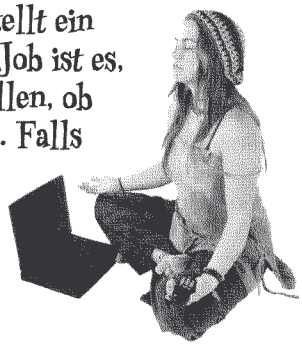
Jetzt mach aber mal 'nen Punkt! DAS stimmt definitiv nicht. Manche statischen Variablen sind für ein System absolut entscheidend. Und selbst die nicht lebensnotwendigen sind sicherlich praktisch.

Warum sagst du das? Und was ist denn verkehrt an statischen Methoden?

Sicher, ich weiß schon, dass Objekte im Mittelpunkt eines OO-Designs stehen sollten – aber nur weil es da draußen ein paar Programmierer gibt, die nicht viel Ahnung haben, solltest du doch das Kind nicht mit dem Bytecode ausschütten. Es gibt Zeit und Raum genug, auch für statische Elemente, und wenn du mal eins brauchst, ist es unschlagbar.

# Spielen Sie Compiler

Die Java-Datei auf dieser Seite stellt ein vollständiges Programm dar. Ihr Job ist es, Compiler zu spielen und festzustellen, ob sich diese Datei kompilieren lässt. Falls nein - wie könnten Sie das in Ordnung bringen? Und falls ja - was käme heraus?



```
class StaticSuper{

  static {
    System.out.println("statischer Block in super");
  }

  StaticSuper{
    System.out.println("Superkonstruktor");
  }
}

public class StaticTests extends StaticSuper {
  static int zufallszahl;

  static {
    zufallszahl = (int) (Math.random() * 6);
    System.out.println("statischer Block "
                       + zufallszahl);
  }

  StaticTests() {
    System.out.println("Konstruktor");
  }

  public static void main(String [] args) {
    System.out.println("in main-Methode");
    StaticTests st = new StaticTests();
  }
}
```

Falls das kompilierbar ist – welche Ausgabe erhält man?

**Mögliche Ausgabe**

```
Datei Bearbeiten Fenster Hilfe ElektrischAufgeladen
%java StaticTests
statischer Block 4
in main-Methode
statischer Block in super
Superkonstruktor
Konstruktor
```

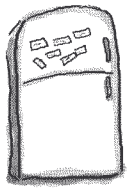
**Mögliche Ausgabe**

```
Datei Bearbeiten Fenster Hilfe Schlag
%java StaticTests
statischer Block in super
statischer Block 3
in main-Methode
Superkonstruktor
Konstruktor
```

In diesem Kapitel haben wir die wunderbare statische Welt von Java erkundet. Entscheiden Sie, ob die folgenden Aussagen dazu jeweils wahr oder falsch sind.

# Wahr oder Falsch?

1. Wenn man die Klasse Math benutzen will, muss man als Erstes eine Instanz davon machen.
2. Sie können einen Konstruktor mit dem Schlüsselwort **`static`** kennzeichnen.
3. Statische Methoden haben keinen Zugriff auf die Instanzvariablen eines Objekts.
4. Es ist guter Stil, eine statische Methode mit Hilfe einer Referenzvariablen aufzurufen.
5. Mit Hilfe von statischen Variablen könnte man die Instanzen einer Klasse zählen.
6. Konstruktoren werden aufgerufen, bevor statische Variablen initialisiert werden.
7. MAX_SIZE wäre ein guter Name für eine statische finale Variable.
8. Ein statischer Initialisiererblock wird noch vor dem Konstruktor einer Klasse ausgeführt.
9. Wenn eine Klasse als final gekennzeichnet ist, müssen alle ihre Methoden als final gekennzeichnet sein.
10. Eine finale Methode kann nur überschrieben werden, wenn ihre Klasse erweitert wird.
11. Es gibt keine Wrapper-Klasse für primitive Boolesche Werte.
12. Ein Wrapper wird benutzt, wenn Sie einen primitiven Typ wie ein Objekt behandeln wollen.
13. Alle parseXxx-Methoden geben einen String zurück.
14. Die Formatierungsklassen (die von E/A entkoppelt sind) gehören zum Paket java.format.



# Mondcode-Magneten

Das hier könnte Ihnen tatsächlich nützlich sein! Zusätzlich zu dem, was Sie auf den letzten Seiten über die Manipulation von Datumsangaben gelernt haben, brauchen Sie noch ein paar weitere Informationen ... Erstens: Vollmond ist alle 29,52 Tage oder so. Zweitens: Am 7. Januar 2004 war Vollmond. Sie sollen die Codeschnipsel so wieder zusammensetzen, dass sich ein funktionierendes Java-Programm ergibt, mit dem man die unten abgedruckte Ausgabe (plus weitere Vollmonddatumswerte) erhält. (Möglicherweise brauchen Sie nicht alle Magneten. Und fügen Sie alle geschweiften Klammern ein, die nötig sind.) Ach, übrigens – wenn Sie nicht in der zentraleuropäischen Zeitzone leben, sieht Ihre Ausgabe etwas anders aus.

```
long tag1 = c.getTimeInMillis();
c.set(2004,1,7,15,40);
import static java.lang.System.out;
static int tag_IM = 60 * 60 * 24;
("Vollmond am %tc", c));
(c.format
Calendar c = new Calendar();
class Vollmonde {
public static void main(String [] args) {
tag1 += (TAG_IM * 29.52);
for (int x = 0; x < 60; x++) {
static int TAG_IM = 1000 * 60 * 60 * 24;
println
("Vollmond am %t", c));
import java.io.*;
import java.util.*;
static import java.lang.System.out;
c.set(2004,0,7,15,40);
out.println
c.setTimeInMillis(tag1);
(String.format
Calendar c = Calendar.getInstance();
```

```
Datei Bearbeiten Fenster Hilfe Heulen
%java Vollmonde
Vollmond am Fr Feb 06 04:09:01 CET 2004
Vollmond am Sa Mrz 06 16:37:49 CET 2004
Vollmond am Mo Apr 05 06:06:37 CEST 2004
```

# Lösungen zu den Übungen

## Spielen Sie Compiler

```
StaticSuper( ) {
  System.out.println(
    "Superkonstruktor");
}
```

StaticSuper ist ein Konstruktor, in dessen Signatur Klammern () stehen müssen. Wie die unten stehende Ausgabe zeigt, werden die statischen Blöcke von beiden Klassen ausgeführt, noch bevor irgendeiner der beiden Konstruktoren zum Zuge kommt.

**Ausgabe**

```
Datei Bearbeiten Fenster Hilfe Schlag
%java StaticTests
statischer Block in super
statischer Block 3
in main-Methode
Superkonstruktor
Konstruktor
```

## Wahr oder Falsch?

1. Wenn man die Klasse Math benutzen will, muss man als Erstes eine Instanz davon machen. **Falsch**
2. Sie können einen Konstruktor mit dem Schlüsselwort »static« kennzeichnen. **Falsch**
3. Statische Methoden haben keinen Zugriff auf die Instanzvariablen eines Objekts. **Wahr**
4. Es ist guter Stil, eine statische Methode mit Hilfe einer Referenzvariablen aufzurufen. **Falsch**
5. Mit Hilfe von statischen Variablen könnte man die Instanzen einer Klasse zählen. **Wahr**
6. Konstruktoren werden aufgerufen, bevor statische Variablen initialisiert werden. **Falsch**
7. MAX_SIZE wäre ein guter Name für eine statische finale Variable. **Wahr**
8. Ein statischer Initialisiererblock wird noch vor dem Konstruktor einer Klasse ausgeführt. **Wahr**
9. Wenn eine Klasse als final gekennzeichnet ist, müssen alle ihre Methoden als final gekennzeichnet sein. **Falsch**
10. Eine finale Methode kann nur überschrieben werden, wenn ihre Klasse erweitert wird. **Falsch**
11. Es gibt keine Wrapper-Klasse für primitive Boolesche Werte. **Falsch**
12. Ein Wrapper wird benutzt, wenn Sie einen primitiven Typ wie ein Objekt behandeln wollen. **Wahr**
13. Alle parseXxx-Methoden geben einen String zurück. **Falsch**
14. Die Formatierungsklassen (die von E/A entkoppelt sind) gehören zum Paket java.format. **Falsch**

# Lösungen zu den Übungen

```
import java.util.*;

import static java.lang.System.out;

class Vollmonde {

  static int TAG_IM = 1000 * 60 * 60 * 24;

  public static void main(String [] args) {

    Calendar c = Calendar.getInstance();

    c.set(2004,0,7,15,40);

    long tag1 = c.getTimeInMillis();

    for (int x = 0; x < 60; x++) {

      tag1 += (TAG_IM * 29.52);

      c.setTimeInMillis(tag1);

      out.println(String.format("Vollmond am %tc", c));

    }

  }

}
```

Anmerkungen zu den Mondcode-Magneten:

Sie werden vielleicht feststellen, dass ein paar der von diesem Programm gelieferten Datumswerte um einen Tag daneben liegen. Dieser Astronomiekram ist ein bisschen knifflig, und wenn wir ihn perfekt gemacht hätten, wäre er zu komplex und würde sich deshalb hier als Übung nicht mehr eignen.

Hinweis: Eines der Probleme, das Sie versuchen könnten zu lösen, hängt mit den unterschiedlichen Zeitzonen zusammen. Sehen Sie, wo das Problem liegt?

```
Datei Bearbeiten Fenster Hilfe Heulen
%java Vollmond
Vollmond am Fr Feb 06 04:09:01 CET 2004
Vollmond am Sa Mrz 06 16:37:49 CET 2004
Vollmond am Mo Apr 05 06:06:37 CEST 2004
```

# Riskantes Verhalten

## Manches passiert einfach. Die Datei ist nicht da. Der Server läuft nicht.

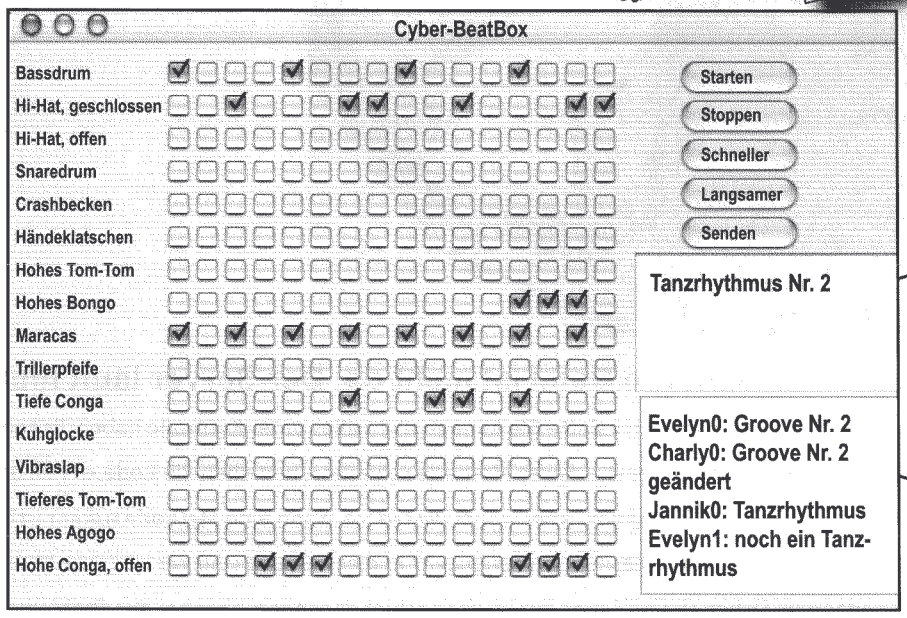
Und wenn Sie ein noch so guter Programmierer sind – Sie können nicht alles unter Kontrolle haben. Es kann etwas schief gehen. *Richtig* schief. Wenn Sie eine riskante Methode schreiben, brauchen Sie Code, der all das Schlimme, das möglicherweise passiert, behandelt. Aber woher *wissen* Sie, wann eine Methode riskant ist? Und wo platzieren Sie den Code, der die ***Ausnahme***situation *behandelt*? In diesem Buch sind wir *wirkliche* Risiken bisher noch nicht eingegangen. Sicher, es sind Dinge zur Laufzeit schief gegangen, aber bei den Problemen handelte es sich vor allem um Schwachstellen in unserem eigenen Code. Bugs. Und die sollten wir zur Entwicklungszeit in Ordnung bringen. Nein, der Fehlerbehandlungscode, über den wir hier sprechen, bezieht sich auf Code, bei dem Sie *nicht garantieren können*, dass er zur Laufzeit funktioniert. Code, der erwartet, dass die Datei im richtigen Verzeichnis liegt, dass der Server läuft oder dass der Thread weiterschläft. Und wir müssen uns *jetzt* darum kümmern – weil wir in *diesem* Kapitel etwas programmieren wollen, das die riskante JavaSound-API benutzt. Wir wollen nämlich einen MIDI-Musikplayer programmieren.

# Bauen wir uns eine Musikmaschine

Im Verlauf der nächsten drei Kapitel bauen wir uns mehrere Soundanwendungen, einschließlich eines BeatBox-Schlagzeugprogramms. Und noch bevor wir mit dem Buch fertig sind, werden wir eine Version für mehrere Benutzer haben, mit der Sie Ihre Schlagzeugkompositionen an einen anderen Teilnehmer schicken können, ähnlich wie in einem Chatroom. Sie werden das Ding komplett selbst schreiben; für die GUI-Teile können Sie allerdings wahlweise auch fertig vorbereiteten Code verwenden. Okay, es braucht nicht gerade jede IT-Abteilung einen neuen BeatBox-Server – aber wir tun das ja, um *mehr über Java zu lernen.* Beim Programmieren einer BeatBox können wir nämlich *beim Lernen* gleichzeitig Spaß haben.

## Die fertige BeatBox sieht etwa so aus wie das hier:

Sie setzen Haken in die Kästchen und erstellen damit ein in einer Schleife sich wiederholendes Schlagzeugmotiv (ein Muster mit 16 Schlägen, auch als >>Drum-Pattern<< bezeichnet).




Ihre Nachricht, die gemeinsam mit Ihrem aktuellen Drum-Pattern an die anderen Benutzer geschickt wird, wenn Sie auf >>Senden<< klicken.

Eingehende Nachrichten von anderen Teilnehmern. Klicken Sie auf eine davon, um das zugehörige Pattern zu laden, und klicken Sie dann auf >>Starten<<, um es abzuspielen.

Setzen Sie für jeden der 16 »Schläge« Haken in die Kästchen. So ertönen beispielsweise bei Schlag 1 (von 16) die Basstrommel und die Maracas, bei Schlag 2 nichts und bei Schlag 3 die Maracas und das geschlossene Hi-Hat ... Sie verstehen, wie es funktioniert? Wenn Sie auf »Starten« klicken, wird Ihr Pattern in einer Schleife so lange gespielt, bis Sie »Stoppen« drücken. Sie können jederzeit eins Ihrer eigenen Patterns »einfangen«, indem Sie es an den BeatBox-Server schicken (was bedeutet, dass die anderen Teilnehmer es sich anhören können). Sie können auch jedes der dort eingegangenen Patterns laden, indem Sie auf die zugehörige Nachricht klicken.

# Beginnen wir mit den Grundlagen

Ganz offensichtlich müssen wir noch ein paar Dinge lernen, bevor das Programm fertig wird, z.B. wie man eine Swing-Benutzeroberfläche (eine GUI) baut, wie man übers Netz eine Verbindung zu einem anderen Rechner herstellt, und ein bisschen Eingabe und Ausgabe (E/A), damit wir dem anderen Rechner auch etwas schicken können.

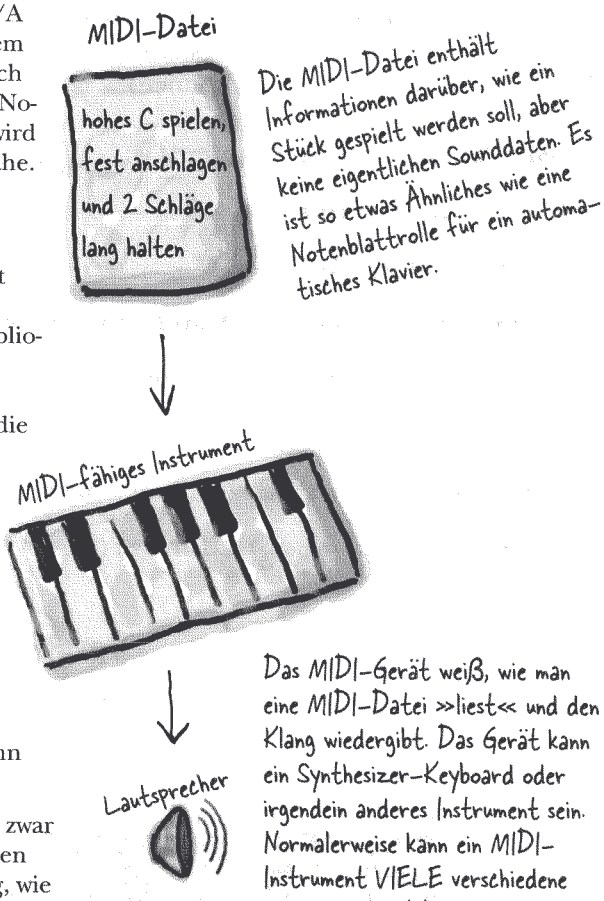
Ach ja, und die JavaSound-API. *Damit* machen wir in diesem Kapitel den Anfang. Vorläufig können Sie GUI, Arbeiten im Netz und E/A wieder vergessen und sich nur darauf konzentrieren, wie Sie Ihrem Computer MIDI-erzeugten Sound entlocken. Und machen Sie sich keine Sorgen, falls Sie keine Ahnung von MIDI haben oder vom Notenlesen oder vom Musikmachen. Alles, was Sie lernen müssen, wird hier besprochen. Ihr Plattenvertrag ist schon fast zum Greifen nahe.

## Die JavaSound-API

JavaSound ist eine Sammlung von Klassen und Interfaces, die mit Version 1.3 zu Java hinzugefügt wurden. Es sind keine speziellen Erweiterungen, sondern sie gehören zur J2SE-Standardklassenbibliothek. JavaSound besteht aus zwei Teilen, MIDI und Sampled. In diesem Buch verwenden wir nur MIDI (abgekürzt für Musical Instrument Digital Interface); das ist ein Standardprotokoll, das die Kommunikation zwischen verschiedenen Arten von Geräten für elektronische Musik ermöglicht. Für unsere BeatBox-Anwendung können Sie sich MIDI aber einfach als *eine Art Notenblatt* vorstellen, das Sie in ein Gerät stecken, das Sie sich wiederum ähnlich wie ein automatisches Hightech-Klavier vorstellen können. Anders ausgedrückt: MIDI-Daten enthalten eigentlich gar keinen *Sound*, sondern die *Anweisungen*, nach denen ein MIDI-lesendes Instrument Klänge wiedergeben kann. Und hier noch eine andere Analogie: Stellen Sie sich eine MIDI-Datei wie ein HTML-Dokument vor – das Instrument, das die MIDI-Datei ausgibt (d.h. sie *abspielt*), entspricht dann dem Webbrowser.

MIDI-Daten sagen, *was* zu tun ist (eingestrichenes C spielen, und zwar soundso fest anschlagen, soundso lange halten usw.), aber sie sagen nichts darüber, welchen *Klang* man hört. MIDI hat keine Ahnung, wie man einen Flöten- oder Klavierklang oder den Sound einer Jimi-Hendrix-Gitarre erzeugt. Für den tatsächlichen Sound brauchen wir ein Instrument (ein MIDI-Gerät), das eine MIDI-Datei lesen und abspielen kann. Das Gerät ähnelt allerdings in der Regel eher *einer ganzen Band oder einem Orchester* von Instrumenten. Und das Instrument kann auch ein reales Gerät sein, wie die von Rockmusikern gespielten elektronischen Keyboard-Synthesizer, oder gar ein Instrument, das vollständig aus Software besteht und nur auf unserem Rechner existiert.

Für unsere BeatBox verwenden wir nur das eingebaute »Software-only«-Instrument, das mit Java geliefert wird. Man bezeichnet es als *Synthesizer* (manche nennen es auch Software-Synth), weil es synthetische Klänge *produziert*. *Hörbare* Klänge.



# Erst einmal brauchen wir einen Sequencer

Bevor wir überhaupt irgendwelche Klänge abspielen können, brauchen wir ein Sequencer-Objekt; das ist das Objekt, das alle MIDI-Daten entgegennimmt und sie an die richtigen Instrumente sendet. Das Ding, das die Musik *spielt*. Ein Sequencer kann viele verschiedene Dinge, aber in diesem Buch setzen wir ihn ausschließlich als Wiedergabegerät ein. Wie ein CD-Player in Ihrer Stereoanlage, aber mit ein paar zusätzlichen Funktionen. Die Klasse Sequencer gehört zum Paket javax.sound.midi (Teil der Java-Standardbibliothek seit Version 1.3). Vergewissern wir uns also als Erstes, dass wir ein Sequencer-Objekt erzeugen können (bzw. uns geben lassen können):

```
import javax.sound.midi.*;

public class MusikTest1 {

  public void spielen() {

    Sequencer sequencer = MidiSystem.getSequencer();

    System.out.println("Wir haben einen Sequencer.");

  } // spielen schließen

  public static void main(String[] args){

    MusikTest1 mt = new MusikTest1();

    mt.spielen();

  } // main schließen

} // Klasse schließen
```

Importieren Sie das Paket javax.sound.midi.

Wir brauchen ein Sequencer-Objekt. Es ist der wichtigste Teil des MIDI-Geräts/-Instruments, das wir verwenden, und sorgt dafür, dass alle MIDI-Informationen zu einem »Musikstück« – na was wohl sonst – sequenziert werden. Wir erzeugen aber selbst kein neues Sequencer-Objekt, sondern müssen uns eins vom MIDI-System geben lassen.

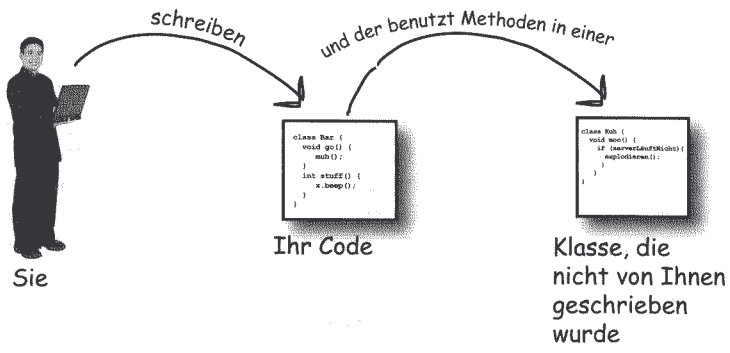
## Da stimmt etwas nicht!

Dieser Code lässt sich nicht kompilieren! Der Compiler sagt, da ist eine »nicht gemeldete Ausnahme«, die abgefangen oder deklariert werden muss.

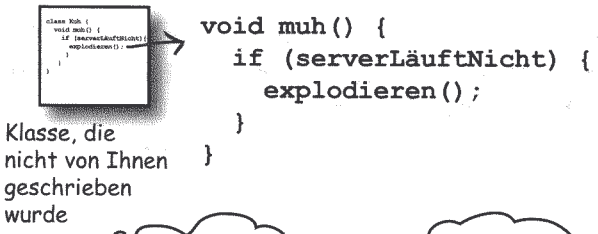
```
Datei Bearbeiten Fenster Hilfe WieBitte?

% javac MusikTest1.java
MusikTest1.java:13: unreported exception javax.sound.midi.
MidiUnavailableException; must be caught or declared to be
thrown
    Sequencer sequencer = MidiSystem.getSequencer();
                                     ^
1 error
```

# Was passiert, wenn eine Methode, die Sie aufrufen wollen (wahrscheinlich in einer nicht selbst geschriebenen Klasse), riskant ist?

**1** **Nehmen wir an, Sie wollen eine Methode in einer Klasse aufrufen, die Sie nicht selbst geschrieben haben.**

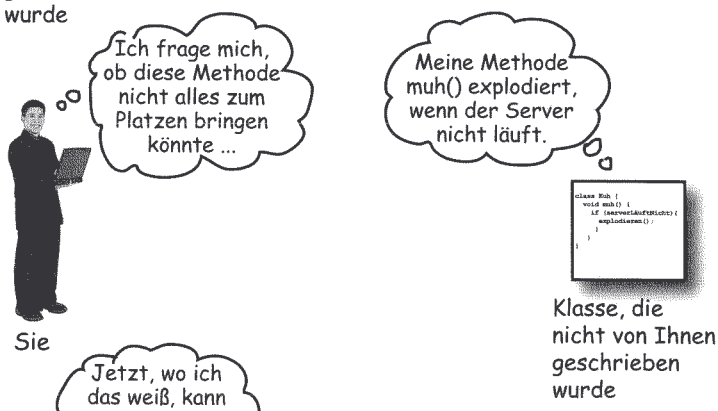



**2** **Diese Methode macht etwas Riskantes – etwas, das zur Laufzeit vielleicht nicht funktioniert.**




```
void muh() {
  if (serverLäuftNicht) {
    explodieren();
  }
}
```

**3** **Es muss Ihnen *bekannt* sein, dass die aufgerufene Methode riskant ist.**




**4** **Dann schreiben Sie Code, der den Fehler behandeln kann, falls er *tatsächlich* eintritt. Sie müssen vorbereitet sein – einfach nur »für den Fall, dass ...«.**

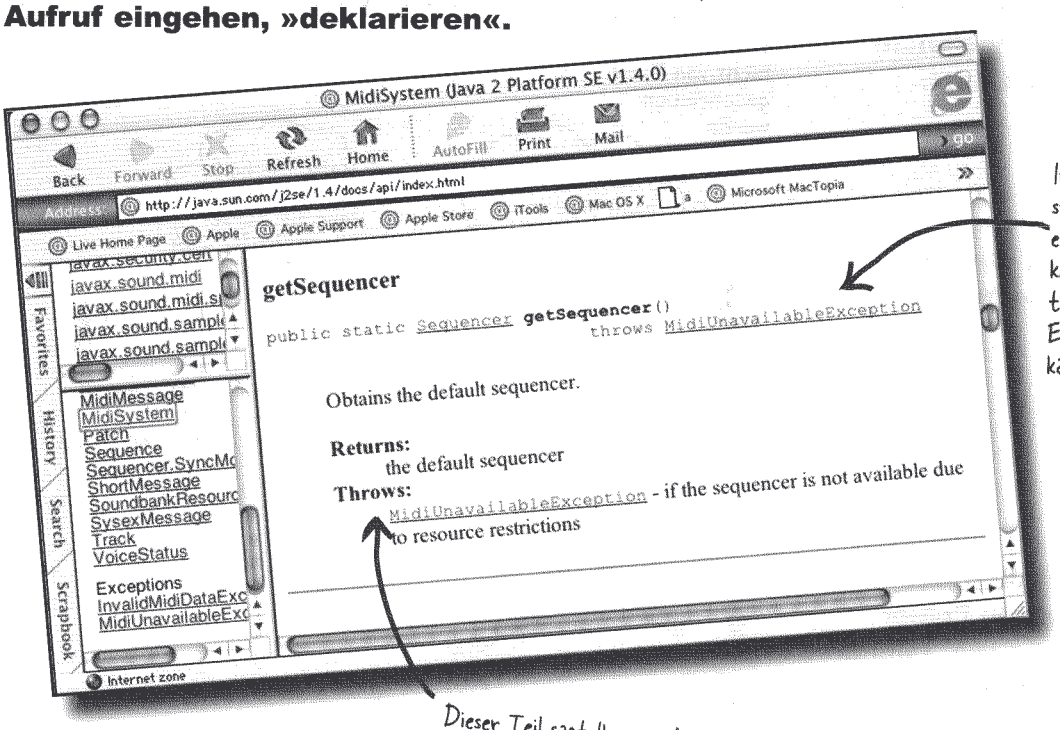
# Mit Hilfe von *Exceptions* teilen Java-Methoden dem aufrufenden Code mit:

## »Es ist etwas Schlimmes passiert. Ich bin gescheitert.«

Der Exception-Handling-Mechanismus von Java ist ein sauberer und sicherer Weg zur Behandlung von »Ausnahmesituationen«, die zur Laufzeit eintreten; Sie können damit Ihren gesamten Fehlerbehandlungscode leicht lesbar an *einer* Stelle unterbringen. Dazu muss Ihnen allerdings vorher *bekannt* sein, dass die von Ihnen aufgerufene Methode riskant ist (d.h., dass sie eine Exception auslösen könnte), damit Sie Code schreiben können, der mit dieser Situation fertig wird. Wenn Sie *wissen, dass Sie* beim Aufruf einer bestimmten Methode möglicherweise eine Exception erhalten, können Sie auf das Problem, das diese Ausnahme ausgelöst hat, *vorbereitet* sein und sich vielleicht sogar davon *erholen*.

Woher Sie *wissen*, ob eine Methode eine Exception auslöst? Sie finden in der Deklaration der riskanten Methode eine **`throws`**-Klausel (d.h. die Angabe, dass eine Exception »geworfen« wird).

**Die Methode `getSequencer()` birgt ein Risiko. Sie kann zur Laufzeit scheitern. Deshalb muss sie das Risiko, das *Sie* bei ihrem Aufruf eingehen, »deklarieren«.**



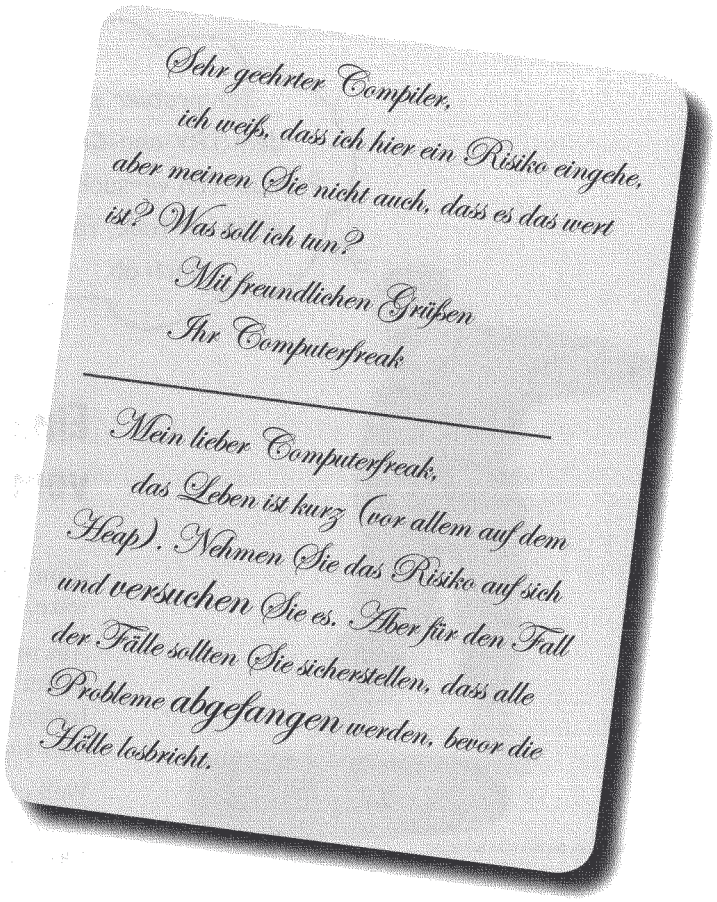

In der API-Dokumentation steht, dass getSequencer() eine Exception »werfen« kann: MidiUnavailableException. Eine Methode muss die Exceptions, die sie auslösen könnte, deklarieren.

Dieser Teil sagt Ihnen, WANN Sie die Exception erhalten könnten – in diesem Fall auf Grund der eingeschränkten Verfügbarkeit einer Ressource (was einfach heißen könnte, dass der Sequencer zu dem Zeitpunkt schon anderweitig verwendet wird).

# Der Compiler muss wissen, dass IHNEN bekannt ist: Ich rufe eine riskante Methode auf.

Wenn Sie den riskanten Code in etwas verpacken, das man als **try/catch** bezeichnet, ist der Compiler beruhigt.

Ein try/catch-Block sagt dem Compiler: Sie *wissen*, dass bei der von Ihnen aufgerufenen Methode eine Ausnahmesituation eintreten kann, und Sie sind darauf vorbereitet, diese Exception zu behandeln. W*ie* Sie das machen, ist dem Compiler egal – ihn interessiert lediglich, dass Sie sagen, Sie kümmern sich darum.




```
import javax.sound.midi.*;

public class MusikTest1 {
  public void spielen() {

    try {
      Sequencer sequencer = MidiSystem.getSequencer();
      System.out.println("Ja! Wir haben einen Sequencer!");
    } catch (MidiUnavailableException ex) {
          System.out.println("So ein Mist!");
    }
  } // spielen schließen

  public static void main(String[] args){
     MusikTest1 mt = new MusikTest1();
     mt.spielen();
  } // main schließen
} // Klasse schließen
```

Stecken Sie die riskante Sache in einen >>try<<-Block.

Machen Sie einen >>catch<<-Block für das, was geschehen soll, wenn die Ausnahmesituation eintritt – mit anderen Worten, wenn der Aufruf von getSequencer() eine MidiUnavailable-Exception auslöst.

Versuchen Sie das nicht zu Hause!

# Eine Exception ist ein Objekt ... vom Typ Exception.

Zum Glück für uns, denn wenn sie vom Typ Brokkoli wäre, könnten wir uns das viel schlechter merken.

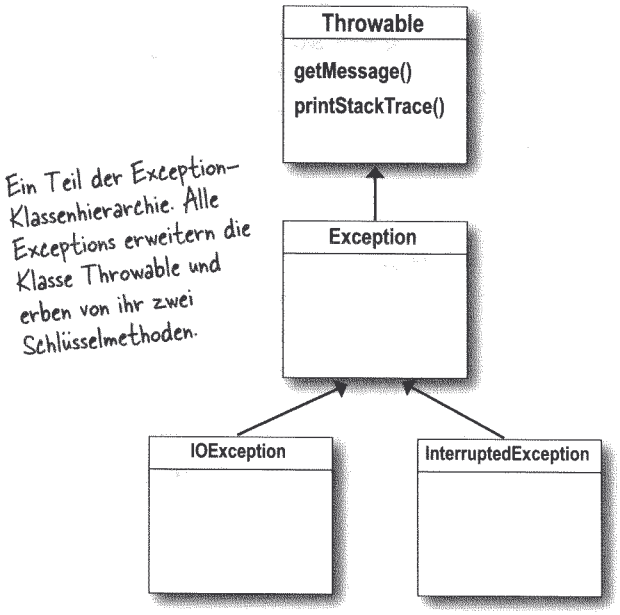
Aus unseren Kapiteln über Polymorphie wissen Sie sicher noch, dass ein Objekt vom Typ Exception eine Instanz einer beliebigen *Unterklasse* von Exception sein *kann*.

Weil eine *Exception* ein Objekt ist, ist das, was Sie *abfangen*, ein Objekt. Im folgenden Code wird das **catch**-Argument mit dem Typ Exception deklariert, und die Parameter-Referenzvariable heißt *ex*.

```
try {
  // tut was Riskantes
} catch (Exception ex) {
  // versucht, noch etwas zu retten
  // oder wiederherzustellen
}
```

Genau wie die Deklaration eines Methodenarguments.

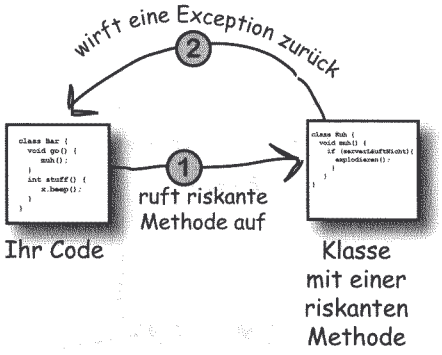
Dieser Code wird nur dann ausgeführt, wenn eine Exception ausgelöst wird.




Ein Teil der Exception-Klassenhierarchie. Alle Exceptions erweitern die Klasse Throwable und erben von ihr zwei Schlüsselmethoden.

Was Sie in einen catch-Block schreiben, hängt davon ab, welche Exception ausgelöst wurde. Wenn beispielsweise ein Server nicht läuft, könnten Sie den catch-Block dazu benutzen, es mit einem anderen Server zu versuchen. Ist eine Datei nicht da, könnten Sie den Benutzer um Hilfe bei der Suche bitten.

# Wenn es *Ihr* Code ist, der die Exception abfängt – wessen Code löst sie dann aus?




Sie werden beim Java-Programmieren wesentlich mehr Zeit mit dem *Behandeln* von Exceptions verbringen als damit, diese selbst zu *erzeugen* und *auszulösen*. Vorläufig müssen Sie nur wissen: Wenn Ihr Code eine riskante Methode *aufruft* – d.h. eine Methode, die eine Exception deklariert –, dann ist es die riskante Methode, die die Exception *auslöst* und sie *Ihnen*, dem Aufrufer, »zuwirft«.

Tatsächlich könnten Sie auch beide Klassen selbst geschrieben haben. Es ist wirklich egal, wer den Code schreibt ... wichtig ist nur, dass Sie wissen, welche Methode die Exception *auslöst* und welche Methode sie *abfängt*.

Wer Code schreibt, der eine Exception auslösen könnte, muss die Exception *deklarieren*.

**1 Riskanter Code, der eine Exception auslöst:**

Diese Methode MUSS der Welt mitteilen (mittels Deklaration), dass sie eine SchlimmeException auslöst.

```
public void risikoEingehen() throws SchlimmeException {
  if (alleHoffnungVerloren) {
    throw new SchlimmeException();
  }
}
```

Ein Exception-Objekt erzeugen und werfen.

**2 Ihr eigener Code, der die riskante Methode *aufruft*:**

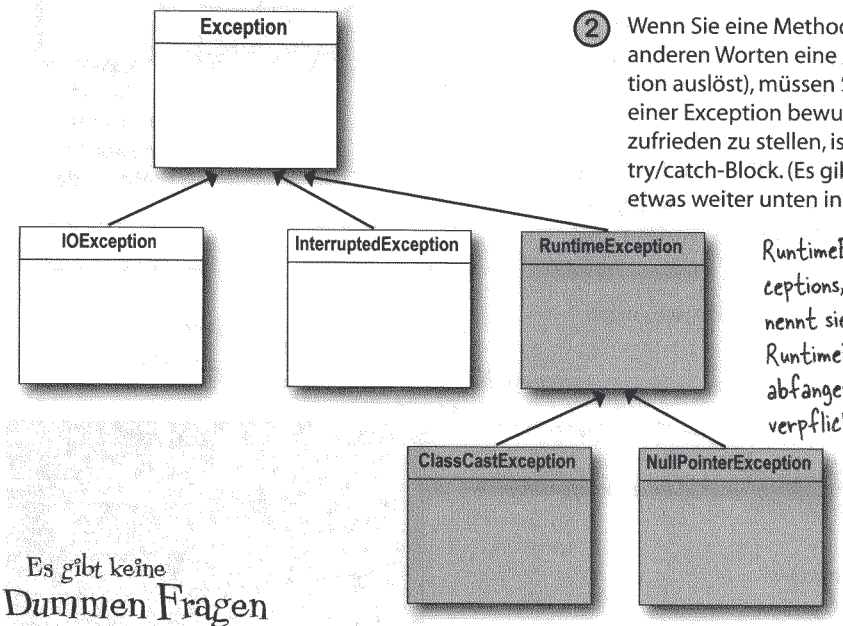
```
public void daumenDrücken() {
  try {
    einObjekt.risikoEingehen();
  } catch (SchlimmeException ex) {
    System.out.println("Aaargh!");
    ex.printStackTrace();
}
```

Sollte nach der Exception nichts mehr zu retten sein, lassen Sie sich WENIGSTENS von der Methode printStackTrace(), die alle Exceptions erben, einen Stacktrace ausgeben (eine Auflistung der aufgerufenen Methoden, anhand derer Sie den Fehler zurückverfolgen können).

Eine Methode fängt ab, was eine andere Methode wirft. Eine Exception wird immer zum Aufrufer zurückgeworfen.

Die werfende Methode muss deklarieren, dass sie die Exception auslösen könnte.

Exceptions, die NICHT Unterklassen von RuntimeException sind, werden vom Compiler geprüft. Diese geprüften Exceptions werden auch mit dem englischen Begriff »checked Exceptions« bezeichnet.




RuntimeExceptions, d.h. zur Laufzeit ausgelöste Exceptions, werden vom Compiler NICHT geprüft. Man nennt sie – na wie wohl – »ungeprüfte Exceptions«. RuntimeExceptions können Sie zwar auch werfen, abfangen und deklarieren, aber Sie sind nicht dazu verpflichtet, und der Compiler prüft das nicht.

## Der Compiler überprüft alles außer Laufzeit-Exceptions.

## Der Compiler garantiert:

1. Wenn Sie in Ihrem Code eine Exception *auslösen*, *müssen* Sie diese in Ihrem Methodenkopf mit dem Schlüsselwort *throws* deklarieren.

2. Wenn Sie eine Methode *aufrufen*, die eine Exception auslöst (mit anderen Worten eine Methode, die *deklariert*, dass sie eine Exception auslöst), müssen Sie *bestätigen*, dass Sie sich der Möglichkeit einer Exception bewusst sind. Eine Möglichkeit, den Compiler zufrieden zu stellen, ist das Einschließen des Aufrufs in einen try/catch-Block. (Es gibt noch eine zweite Möglichkeit, die wir uns etwas weiter unten in diesem Kapitel noch ansehen.)

## Es gibt keine Dummen Fragen

**F: Moment mal! Wie kommt es, dass wir hier ERSTMALS einer Exception mit try/catch begegnen mussten? Was ist mit den Exceptions, die ich schon hatte, z.B. NullPointerException und die Exception bei einer Division durch null? Ich habe mir sogar eine NumberFormatException von der Methode Integer.parseInt() eingehandelt. Warum mussten wir die denn nicht abfangen?**

A: Der Compiler kümmert sich um alle Unterklassen von Exception, *außer* wenn diese zu einem speziellen Typ gehören – RuntimeException. Alle Exceptions, die RuntimeException erweitern, werden unbeanstandet durchgelassen. Solche Laufzeit-Exceptions können überall ausgelöst werden, ob mit oder ohne Deklaration oder try/catch-Block. Der Compiler macht sich nicht die Mühe zu überprüfen, ob eine Methode eine RuntimeException deklariert oder ob der Aufrufer sich darüber im Klaren ist, dass er zur Laufzeit diese Exception erhalten könnte.

**F: Jetzt will ich's genau wissen. WARUM sind dem Compiler diese RuntimeExceptions egal? Können nicht auch sie möglicherweise die ganze Vorstellung platzen lassen?**

A: Die meisten RuntimeExceptions sind auf logische Probleme in Ihrem Code zurückzuführen und nicht darauf, dass irgendeine Bedingung zur Laufzeit in einer Art und Weise nicht erfüllt ist, die Sie weder vorhersagen noch verhindern können. Sie *können nicht* garantieren, dass die Datei da ist. Sie *können nicht* garantieren, dass der Server läuft. Aber Sie *können* sicherstellen, dass Ihr Code nicht das Ende eines Arrays überschreiten will (denn dafür gibt es das Attribut `.length`).

Zur Entwicklungs- und zur Testzeit SOLLEN RuntimeExceptions ausgelöst werden. Man sollte z.B. nicht etwas in einem try/catch codieren – mit all dem damit verbundenen Mehraufwand –, um etwas abzufangen, was an sich gar nicht passieren sollte.

Ein try/catch ist zur Behandlung von Ausnahmesituationen da, nicht für Schwachstellen in Ihrem Code. Benutzen Sie Ihre catch-Blöcke, um sich aus Situationen zu retten, für deren Erfolg Sie nicht garantieren können. Oder geben Sie wenigstens eine Meldung für den Benutzer aus und einen Stacktrace, damit irgendjemand herausfinden kann, was passiert ist.

## Punkt für Punkt

- Eine Methode kann eine Exception auslösen (»werfen«), wenn etwas zur Laufzeit fehlschlägt.
- Eine Exception oder Ausnahme ist immer ein Objekt vom Typ Exception. (Erinnern Sie sich an die Kapitel über Polymorphie: Das bedeutet, dass es ein Objekt einer Klasse ist, in deren Vererbungsbaum irgendwo weiter oben Exception steht.)
- Exceptions vom Typ **RuntimeException** schenkt der Compiler KEINE Beachtung. Eine solche zur Laufzeit auftretende Exception muss weder deklariert noch mit einem try/catch-Block umgeben werden (auch wenn Sie beides tun dürfen, einzeln oder kombiniert).
- Alle Exceptions, um die sich der Compiler kümmert, werden als »geprüfte Exceptions« bezeichnet, was eigentlich *Compiler*-geprüfte Exceptions bedeutet. Nur RuntimeExceptions sind von der Prüfung durch den Compiler ausgenommen. Alle anderen Exceptions müssen in Ihrem Code den Regeln entsprechend von Ihnen berücksichtigt werden.
- Eine Methode »wirft« eine Exception mit dem Schlüsselwort **`throw`**, gefolgt von einem neuen Exception-Objekt:

```
throw new KoffeinMangelException();
```

- Methoden, die eine geprüfte Exception auslösen *könnten*, ***müssen*** das ankündigen, indem Sie **`throws Exception`** deklarieren.
- Wenn Ihr Code eine Methode aufruft, die eine geprüfte Exception auslöst, muss er dem Compiler zusichern, dass er Vorsorgemaßnahmen getroffen hat.
- Wenn Sie bereit sind, die Exception zu behandeln, umgeben Sie den Aufruf mit try/catch und schreiben Ihren Code für das Exception-Handling und die Wiederherstellungsmaßnahmen in den catch-Block.
- Wenn Sie nicht bereit sind, die Exception zu behandeln, können Sie den Compiler trotzdem glücklich machen, indem Sie der Exception offiziell »ausweichen«. Wie man das macht, besprechen wir etwas weiter unten in diesem Kapitel.

### LernTipp

Möchten Sie etwas Neues lernen, richten Sie es so ein, dass es das *Letzte* vor dem Schlafengehen ist, das Sie versuchen zu lernen. Wenn Sie dieses Buch also aus der Hand gelegt haben (vorausgesetzt, Sie können sich davon losreißen), lesen Sie danach nichts Spannenderes mehr als die Rückseite einer Cornflakes-Packung. Ihr Gehirn braucht Zeit, um zu verarbeiten, was Sie gelesen und gelernt haben. Das könnte mehrere *Stunden* dauern. Sollten Sie versuchen, etwas Neues direkt auf Ihr Java zu packen, könnte ein Teil des Java-Stoffs nicht »hängen bleiben«.

Das betrifft natürlich nicht das Lernen einer körperlichen Fertigkeit. Wenn Sie Ihre neueste Kickbox-Übung für die Meisterschaft trainieren, wird Ihr Java-Lernen davon sicher nicht beeinträchtigt.

Um die besten Ergebnisse zu erhalten, sollten Sie dieses Buch direkt vor dem Schlafengehen lesen (oder sich wenigstens die Bilder anschauen).

### Spitzen Sie Ihren Bleistift

**Welcher von diesen Vorgängen löst Ihrer Meinung nach eine Exception aus, um die sich der Compiler kümmern würde? Wir interessieren uns nur für Sachen, über die Sie in Ihrem Code keine Kontrolle haben. Den ersten Haken haben wir schon gesetzt.**

(Weil es der einfachste war.)

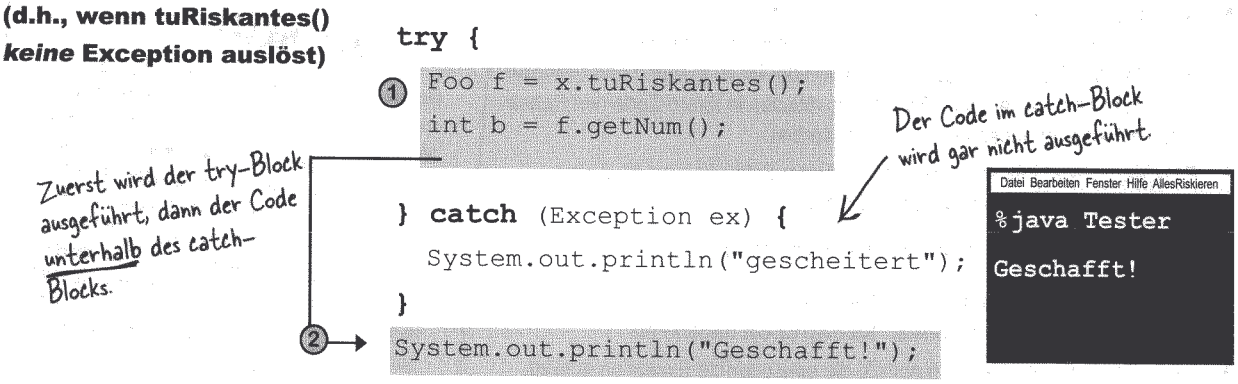
| Was Sie tun möchten | Was schief gehen könnte |
|---|---|
| ✓ eine Verbindung zu einem Remote-Server herstellen | der Server läuft nicht |
| __ beim Zugriff auf ein Array dessen Länge überschreiten | |
| __ ein Fenster auf dem Bildschirm anzeigen | |
| __ Daten aus einer Datenbank abrufen | |
| __ nachsehen, ob eine Textdatei dort liegt, wo Sie sie *vermuten* | |
| __ eine neue Datei erzeugen | |
| __ ein Zeichen von der Kommandozeile einlesen | |

# Flusskontrolle in try/catch-Blöcken

Wenn Sie eine riskante Methode aufrufen, gibt es zwei Möglichkeiten: Entweder ist die riskante Methode erfolgreich und der try-Block wird abgeschlossen, oder die riskante Methode löst eine Exception aus (und »wirft« sie Ihrer aufrufenden Methode zu).

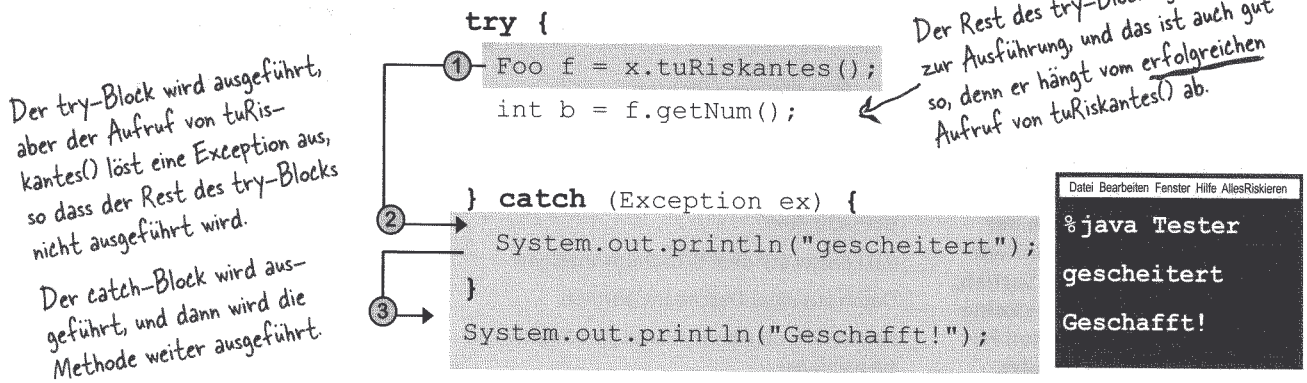
## Wenn »try« erfolgreich ist

**(d.h., wenn tuRiskantes() *keine* Exception auslöst)**




## Wenn »try« scheitert

**(weil tuRiskantes() *eine Exception auslöst*)**



# Und zu guter Letzt finally: für das, was Sie *auf jeden Fall* tun wollen

Wenn Sie etwas zu kochen versuchen, schalten Sie erst einmal den Herd ein.

Wenn das, was Sie ausprobieren, völlig **misslingt,** ***müssen Sie den Herd ausschalten.***

Wenn das, was Sie ausprobieren, **gelingt,** ***müssen Sie den Herd ausschalten.***

***Sie müssen auf jeden Fall den Herd ausschalten – egal, was passiert!***

## In einen finally-Block schreiben Sie Code, der *ungeachtet* einer Exception ausgeführt werden *muss.*

```
try {
  herdEinschalten();
  x.backen();
} catch (BackException ex) {
     ex.printStackTrace();
} finally {
     herdAusschalten();
}
```

Ohne finally müssen Sie herdAusschalten() sowohl in den try- als auch in den catch-Block schreiben, denn ***Sie müssen auf jeden Fall den Herd ausschalten.*** Mit finally können Sie all Ihren wichtigen Code zum Aufräumen an *einer* Stelle unterbringen, statt ihn wie hier doppelt zu schreiben:

```
try {
  herdEinschalten();
  x.backen();
  herdAusschalten();
} catch (BackException ex) {
     ex.printStackTrace();
     herdAusschalten();
}
```

**Wenn der try-Block scheitert (auf Grund einer Exception),** springt die Flusskontrolle sofort zum catch-Block. Ist der catch-Block abgeschlossen, wird der finally-Block ausgeführt. Nach Abschluss des finally-Blocks wird mit dem Rest der Methode fortgefahren.

**Wenn der try-Block erfolgreich ausgeführt wird (*ohne* Exception),** überspringt die Flusskontrolle den catch-Block und geht zum finally-Block. Nach Abschluss des finally-Blocks wird mit dem Rest der Methode fortgefahren.

**Wenn im try- oder catch-Block eine return-Anweisung steht, wird finally dennoch ausgeführt!** Die Flusskontrolle springt zu finally und dann zurück zu return.

Spitzen Sie Ihren Bleistift

# Flusskontrolle

**Sehen Sie sich den links stehenden Code an. Welche Ausgabe würde Ihrer Meinung nach dieses Programm liefern? Und welche Ausgabe bei einer Änderung der dritten Programmzeile in `String test = "ja";`? Gehen Sie davon aus, dass GruselException Exception erweitert.**

```
public class ExceptionTest {

  public static void main(String [] args) {

    String test = "nein";
    try {
      System.out.println("Anfang try-Block");
      tuRiskantes(test);
      System.out.println("Ende try-Block");
    } catch (GruselException ge) {
      System.out.println("gruselige Exception");
    } finally {
      System.out.println("finally");
    }
    System.out.println("Ende von main");
  }

  static void tuRiskantes(String test) throws GruselException {
    System.out.println("Anfang Risiko");
    if ("ja".equals(test)) {
       throw new GruselException();
    }
    System.out.println("Ende Risiko");
    return;
  }
}
```

**Ausgabe mit `test = "nein"`**

**Ausgabe mit `test = "ja"`**

Mit test = »nein«: Anfang try-Block – Anfang Risiko – Ende Risiko – Ende try-Block – finally – Ende von main

Mit test = »ja«: Anfang try-Block – Anfang Risiko – gruselige Exception – finally – Ende von main

# Hatten wir schon erwähnt, dass eine Methode mehr als eine Exception auslösen kann?

Eine Methode kann mehrere Exceptions auslösen, wenn sie das denn unbedingt tun muss. Allerdings muss die Methode *alle* geprüften Exceptions deklarieren, die sie auslösen kann (wenn zwei oder mehr der Exceptions eine gemeinsame Superklasse haben, kann die Methode aber auch einfach die Superklasse deklarieren).

## *Abfangen* von Mehrfach-Exceptions

Der Compiler wird sich vergewissern, dass Sie *alle* geprüften Exceptions, die die von Ihnen aufgerufene Methode auslöst, auch behandelt haben. Reihen Sie die *catch*-Blöcke hintereinander unter dem *try*-Block auf. Manchmal spielt die Reihenfolge der catch-Blöcke dabei eine Rolle, aber dazu kommen wir später noch.

```
public class Wäsche {
  public void waschen() throws HosenException, DessousException {
    // Code, der eine von beiden oder beide
    // Exceptions auslösen könnte
  }
}
```

Diese Methode deklariert ZWEI Exceptions – zählen Sie nach!

---

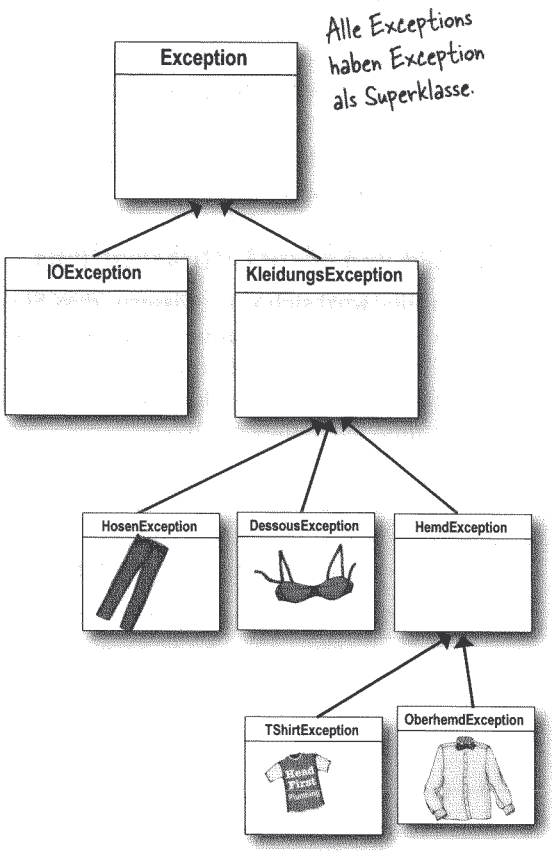
```
public class Foo {
  public void los() {
    Wäsche wäsche = new Wäsche();
    try{
      wäsche.waschen();
    } catch(HosenException hex) {
      // Wiederherstellungscode

    } catch(DessousException dex) {
      // Wiederherstellungscode
    }
  }
}
```

Wenn waschen() eine HosenException wirft, landet diese im catch-Block für HosenExceptions.

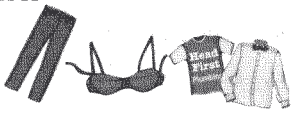
Wenn waschen() eine DessousException wirft, landet diese im catch-Block für DessousExceptions.

# Exceptions sind polymorph

Denken Sie daran, Exceptions sind Objekte. Es ist gar nichts so Besonderes an ihnen, abgesehen davon, dass es sich um Dinge handelt, *die **ausgelöst** werden können.* So können Exceptions wie alle guten Objekte polymorph referenziert werden. Ein DessousException-*Objekt* könnte beispielsweise einer KleidungsException-*Referenz* zugewiesen werden. Eine HosenException könnte einer Exception-Referenz zugewiesen werden. Sie verstehen, worum es geht? Der Vorteil für Exceptions liegt darin, dass eine Methode nicht jede mögliche Exception, die sie auslösen könnte, explizit deklarieren muss, sondern eine Superklasse ihrer Exceptions deklarieren kann. Das Gleiche gilt für catch-Blöcke: Sie müssen nicht für jede einzelne mögliche Exception ein catch schreiben, solange Ihr catch-Block (bzw. Ihre catch-Blöcke) alle ausgelösten Exceptions behandeln kann.




## ① Sie können Exceptions mit einem Supertyp der von Ihnen ausgelösten Exceptions DEKLARIEREN.



```
public void waschen() throws KleidungsException {
```

Wenn Sie eine KleidungsException deklarieren, können Sie jede beliebige Unterklasse von KleidungsException auslösen. Das bedeutet, waschen() kann eine HosenException, eine DessousException, eine TShirtException und eine OberhemdException werfen, ohne diese explizit einzeln zu deklarieren.

## ② Sie können Exceptions mit einem Supertyp der ausgelösten Exception ABFANGEN.

```
try {
  wäsche.waschen();
```

Kann jede beliebige Unterklasse von KleidungsException abfangen.

```
} catch (KleidungsException kex) {
  // Wiederherstellungscode
}
```

```
try {
  wäsche.waschen();
```

Kann nur TShirtException und OberhemdException abfangen.

```
} catch (HemdException hdex) {
  // Wiederherstellungscode
}
```

## Nur weil Sie alles mit einem einzigen polymorphen Supercatch abfangen KÖNNEN, sollten Sie das nicht immer auch tun.

Sie *könnten* sich in Ihrem Exception-Handling-Code auf *einen einzigen* catch-Block beschränken, indem Sie in der catch-Klausel den Supertyp Exception benutzen. So fangen Sie *jede* Ausnahme ab, die ausgelöst werden könnte.

```
try {
  wäsche.waschen();
} catch (Exception ex) {
  // Wiederherstellungscode
}
```

Wiederhergestellt ... ja was denn? Dieser catch-Block fängt JEDE x-beliebige Exception, und deshalb werden Sie auch nicht automatisch erfahren, was schief gegangen ist.

## Schreiben Sie für jede Exception, die in einer für sie typischen Weise behandelt werden muss, einen eigenen catch-Block.

Wenn Ihr Code beispielsweise eine TShirtException anders behandelt als eine Dessous-Exception (oder andere Rettungsmaßnahmen ergreift), schreiben Sie für jede einen eigenen catch-Block. Aber wenn Sie alle anderen Typen von KleidungsException gleichartig behandeln, fügen Sie für die Behandlung der restlichen Exceptions eine KleidungsException hinzu.

```
try {
  wäsche.waschen();

} catch (TShirtException tex) {
  // Wiederherstellung nach TShirtException

} catch (DessousException dex) {
  // Wiederherstellung nach DessousException

} catch (KleidungsException kex) {
  // Wiederherstellung nach allen anderen
}
```

TShirtExceptions und DessousExceptions brauchen unterschiedlichen Rettungscode, deshalb sollten Sie verschiedene catch-Blöcke verwenden.

Alle weiteren KleidungsExceptions werden hier abgefangen.

# Mehrfach-catch-Blöcke müssen von klein nach groß geordnet werden

Hier werden TShirtExceptions abgefangen, aber es passen keine anderen Exceptions.

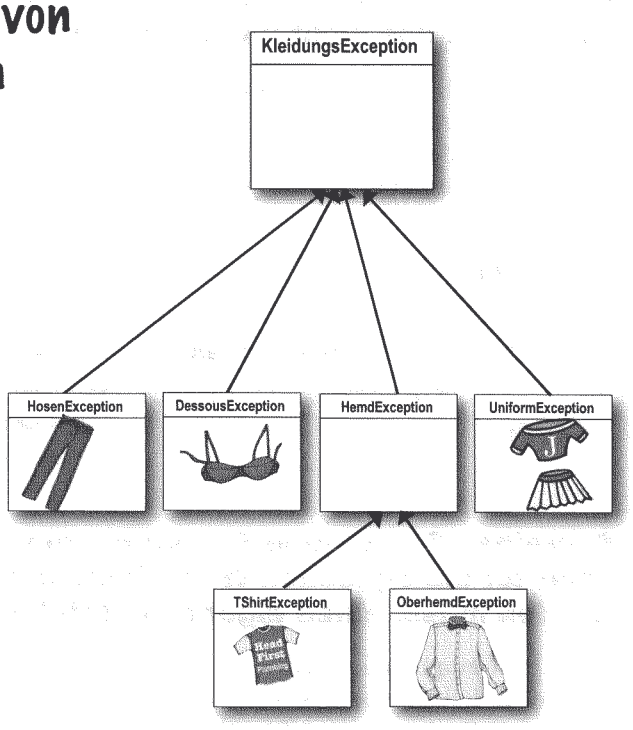
```
catch(TShirtException tex)
```

TShirtExceptions kommen gar nicht erst bis hierher, aber alle anderen Unterklassen von HemdException werden hier abgefangen.

```
catch(HemdException hdex)
```

Hier werden alle KleidungsExceptions abgefangen; TShirtExceptions und HemdExceptions kommen allerdings nicht bis hierher.

```
catch(KleidungsException kex)
```



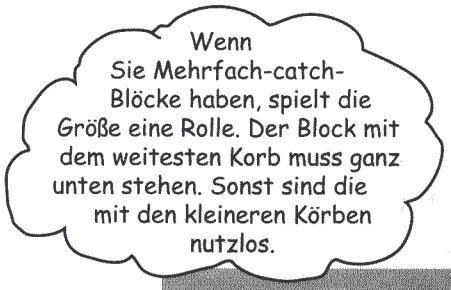

Je weiter oben im Vererbungsbaum, desto größer der »Abfangkorb«. Wenn Sie im Vererbungsbaum in Richtung immer stärker spezialisierter Exception-Klassen nach unten gehen, wird der catch-»Korb« immer kleiner. Das ist nichts weiter als gute alte Polymorphie.

Ein Abfangkorb für eine HemdException ist weit genug für eine TShirtException oder eine OberhemdException (und für jede weitere zukünftige Unterklasse von allem, was HemdException erweitert). Noch größer ist der Korb für KleidungsException (d.h., es gibt noch mehr Dinge, die mit dem Typ KleidungsException referenziert werden können). Er kann – wer hätte das gedacht – eine Exception vom Typ KleidungsException aufnehmen sowie alle Unterklassen davon: HosenException, UniformException, DessousException und HemdException. Die Mutter aller catch-Argumente ist der Typ **Exception.** Damit werden *alle* Exceptions abgefangen, einschließlich (ungeprüfter) Laufzeit-Exceptions; daher werden Sie diesen Typ wohl ausschließlich für Testzwecke verwenden.

# Sie können größere Körbe nicht über kleinere Körbe stellen

Na ja, Sie *können* schon, aber es ist nicht kompilierbar. Catch-Blöcke sind nicht mit überladenen Methoden vergleichbar, bei denen die beste Übereinstimmung herausgesucht wird. Bei catch-Blöcken beginnt die JVM einfach beim ersten und arbeitet sich so lange nach unten vor, bis sie einen findet, der weit genug für die Behandlung dieser Exception ist (anders ausgedrückt: hoch genug im Vererbungsbaum liegt). Wenn Ihr erster catch-Block **`catch(Exception ex)`** ist, weiß der Compiler, es ist sinnlos, noch weitere hinzuzufügen – weil sowieso nichts bis dorthin gelangt.

```
try {
  wäsche.waschen();
} catch (KleidungsException kex) {
  // Wiederherstellung nach KleidungsException
} catch (DessousException dex) {
  // Wiederherstellung nach DessousException
} catch (HemdException hdex) {
  // Wiederherstellung nach HemdException
}
```



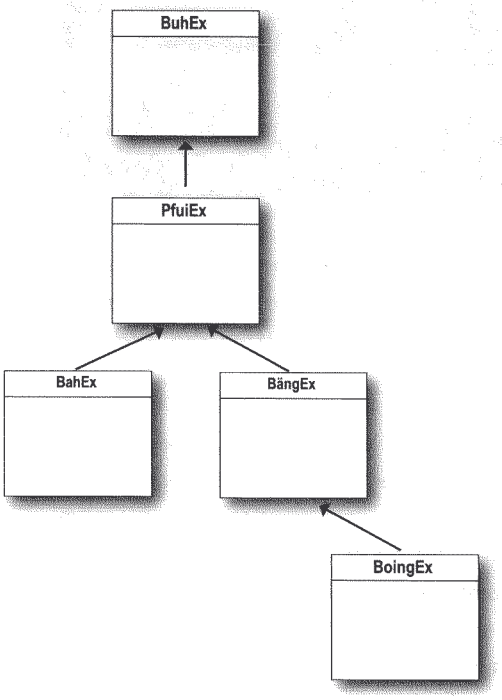
**Gleichrangige können in beliebiger Reihenfolge stehen, denn hier kann nicht einer die Exception des anderen wegfangen.**

Niemand hätte etwas dagegen, wenn Sie HemdException über DessousException stellen. Denn obwohl HemdException ein größerer Typ ist, weil damit auch andere Klassen (die eigenen Unterklassen) abgefangen werden können, kann HemdException doch keine DessousException abfangen – kein Problem also.

Nehmen Sie an, der Code in diesem try/catch-Block ist zulässig. Sie sollen dazu zwei verschiedene Beispiele für Klassendiagramme zeichnen, die die Exception-Klassen exakt repräsentieren. Anders ausgedrückt: Mit welcher Klassenvererbungsstruktur wären die try/catch-Blöcke im Codebeispiel zulässig?

```
try {
  x.tuRiskantes();
} catch (AlphaEx a) {
  // Wiederherstellung nach AlphaEx
} catch (BetaEx b) {
  // Wiederherstellung nach BetaEx
} catch (GammaEx c) {
  // Wiederherstellung nach GammaEx
} catch (DeltaEx d) {
  // Wiederherstellung nach DeltaEx
}
```

---




Ihre Aufgabe ist es, zwei verschiedene *zulässige* try/catch-Strukturen (ähnlich der oben links abgedruckten) zu schreiben, die das links abgebildete Klassendiagramm exakt repräsentieren. Gehen Sie von der Annahme aus, dass ALLE diese Exceptions von der Methode innerhalb des try-Blocks ausgelöst werden können.

# Wenn Sie eine Exception nicht behandeln wollen ...

## ... weichen Sie ihr aus

### Möchten Sie eine Exception nicht behandeln, können Sie ihr ausweichen, indem Sie sie deklarieren.

Wenn Sie eine riskante Methode aufrufen, braucht der Compiler eine Bestätigung darüber, dass Sie davon Kenntnis haben. Meistens heißt das, dass Sie den riskanten Aufruf mit einem try/catch-Block umgeben. Aber Sie haben noch eine Alternative: *Weichen Sie ihr einfach aus* und lassen Sie die Methode, die *Sie* aufgerufen hat, die Exception abfangen.

Das ist ganz einfach: Sie müssen lediglich *deklarieren*, dass *Sie* selbst diese Exception auslösen. Auch wenn *Sie* genau genommen gar nicht der Auslöser sind, aber das spielt keine Rolle. Schließlich sind Sie derjenige, der die Exception an sich vorbeisausen lässt.

Wenn Sie einer Exception ausweichen, haben Sie allerdings keinen try/catch-Block – was passiert also, wenn die riskante Methode (waschen()) *tatsächlich* die Exception wirft?

Wenn eine Methode eine Exception auslöst, fliegt diese Methode sofort vom Stack herunter, und die Exception wird an die Methode weitergegeben, die auf dem Stack direkt darunterliegt – an die *aufrufende Methode*. Wenn aber nun der Aufrufer ebenfalls ausweicht, wird die Exception wieder nicht abgefangen, daher fliegt auch der Aufrufer sofort vom Stack herunter, und die Exception wird an die nächste Methode weitergegeben, und so weiter ... wo führt das hin? Sie werden es gleich noch sehen.

```
public void foo() throws GanzSchlimmeException {
  // riskante Methode ohne try/catch aufrufen
  wäsche.waschen();
}
```

Sie sind nicht der eigentliche Auslöser, aber da Sie kein try/catch für die von Ihnen aufgerufene riskante Methode haben, sind Sie nun selbst »die riskante Methode«. Denn wer auch immer Sie jetzt aufruft, der muss sich mit der Exception befassen.

# Ausweichen mittels Deklarieren schiebt das Unvermeidliche nur auf

**Früher oder später muss sich *irgendjemand* damit befassen. Aber was ist, wenn auch *main()* der Exception ausweicht?**

```
public class WaschMaschine {
  Wäsche wäsche = new Wäsche();

  public void foo() throws KleidungsException {
    wäsche.waschen();
  }

  public static void main (String[] args) throws KleidungsException {
    WaschMaschine a = new WaschMaschine();
    a.foo();
  }
}
```

Beide Methoden weichen der Exception aus (indem Sie sie deklarieren), daher ist niemand da, der sie behandelt! Beim Kompilieren geht alles glatt.

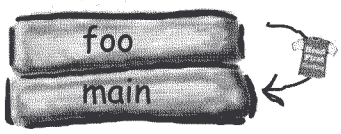
**1** waschen() löst eine KleidungsException aus

main() ruft foo() auf

foo() ruft waschen auf

waschen() wird ausgeführt und löst eine KleidungsException aus

**2** foo() weicht der Exception aus




waschen() fliegt sofort vom Stapel herunter, und die Exception wird an foo() weitergegeben.

Aber foo() hat kein try/catch, also ...

**3** main() weicht der Exception aus

foo() fliegt sofort vom Stapel herunter, und die Exception wird weitergeben ... an wen? Oder was? Es ist ja keiner mehr da außer der JVM, und die denkt sich: »Erwartet nicht, dass ICH euch da raushelfe.«

**4** Die JVM schaltet sich ab.

Als Symbol für eine KleidungsException benutzen wir hier das T-Shirt. Jaja, wir wissen schon ... Sie hätten lieber die Jeans gehabt.

# Behandeln oder deklarieren – so lautet das Gesetz.

**Jetzt haben wir also beide Möglichkeiten dazu kennen gelernt, wie sich der Compiler beim Aufruf einer riskanten Methode (die eine Exception auslöst) zufrieden stellen lässt.**

## 1 BEHANDELN

Umgeben Sie den riskanten Aufruf mit try/catch.

```
try {
  wäsche.waschen();
} catch (KleidungsException kex) {
  // Wiederherstellungscode
}
```

Dieser Abfangtyp sollte besser groß genug sein, um alle Exceptions zu behandeln, die waschen() werfen könnte. Denn sonst wird sich der Compiler nach wie vor darüber beschweren, dass Sie nicht alle Exceptions abfangen.

## 2 DEKLARIEREN (ausweichen)

Deklarieren Sie, dass IHRE Methode die gleichen Exceptions wirft wie die riskante Methode, die Sie aufrufen.

```
public void foo() throws KleidungsException {
  wäsche.waschen();
}
```

Die Methode waschen() löst eine KleidungsException aus, aber die Methode foo() weicht der Exception aus, indem sie diese deklariert. Kein try/catch.

Aber das bedeutet jetzt: Wer immer die Methode foo() aufruft, muss das Gesetz vom Behandeln oder Deklarieren befolgen. Wenn foo() dieser Exception ausweicht (durch Deklaration) und main() foo() aufruft, muss main() sich mit der Exception befassen.

```
public class WaschMaschine {
  Wäsche wäsche = new Wäsche();

  public void foo() throws KleidungsException {
    wäsche.waschen();
  }

  public static void main (String[] args) {
    WaschMaschine a = new WaschMaschine();
    a.foo();
  }
}
```

Das gibt ÄRGER!!

Jetzt lässt sich main() nicht kompilieren; wir erhalten eine Fehlermeldung über eine »unreported exception«. Was den Compiler betrifft, wirft die Methode foo() eine Exception.

Da die Methode foo() der von waschen() ausgelösten KleidungsException ausweicht, muss main() um a.foo() einen try/catch-Block legen – oder main() muss deklarieren, dass sie ebenfalls eine KleidungsException auslöst!

# Zurück zu unserem Musik-Code ...

Auch wenn Sie es inzwischen völlig vergessen haben – zu Beginn dieses Kapitels hatten wir schon einen ersten Blick auf JavaSound-Code geworfen. Wir haben ein Sequencer-Objekt erzeugt, konnten es aber nicht kompilieren, weil die Methode Midi.getSequencer() eine geprüfte Exception deklariert (MidiUnavailableException). Aber jetzt können wir das Problem beheben, indem wir den Aufruf mit einem try/catch-Block umgeben.

```
public void spielen() {
  try {

    Sequencer sequencer = MidiSystem.getSequencer();
    System.out.println("Ja! Wir haben einen Sequencer!");

  } catch (MidiUnavailableException ex) {
    System.out.println("So ein Mist!");
  }
} // spielen schließen
```

Kein Problem mehr mit dem Aufruf von getSequencer(), nachdem wir ihn mit einem try/catch-Block umgeben haben.

Der catch-Parameter muss die >>richtige<< Exception sein. Wenn wir >>catch(FileNotFoundException f)<< schreiben würden, wäre der Code nicht kompilierbar, da eine MidiUnavailableException nicht polymorph in eine FileNotFoundException passen würde.

Denken Sie daran: Ein catch-Block genügt nicht ... Sie müssen das geworfene Ding auch abfangen!

## Exception-Regeln

### 1 Ein catch oder finally ohne try ist nicht erlaubt.

```
void los() {
  Foo f = new Foo();
  f.foof();
  catch(FooException ex) {}
}
```

NICHT ZULÄSSIG! Wo ist der try-Block?

### 2 Sie dürfen keinen Code zwischen try und catch schreiben.

```
try {
  x.tuIrgendwas();
}
int y = 43;
catch(Exception ex) { }
```

NICHT ZULÄSSIG! Sie dürfen keinen Code zwischen try und catch setzen.

### 3 Auf ein try MUSS entweder ein catch oder ein finally folgen.

```
try {
  x.tuIrgendwas();
} finally {
  // aufräumen
}
```

ZULÄSSIG, weil Sie ein finally haben, auch wenn kein catch dasteht. Aber ganz allein darf ein try nicht stehen.

### 4 Bei einem try mit einem finally (aber ohne catch) muss die Exception deklariert werden.

```
void los() throws FooException {
  try {
    x.tuIrgendwas();
  } finally { }
}
```

Ein try ohne catch gehorcht nicht dem Gesetz vom Behandeln oder Deklarieren.

# CodeKüche

Sie müssen es nicht unbedingt selbst machen, aber es macht viel mehr Spaß, wenn Sie es tun.

Der Rest des Kapitels darf übersprungen werden – Sie können auch Fertigcode für all unsere Musikanwendungen nehmen.

Aber wenn Sie mehr über JavaSound lernen wollen, blättern Sie um auf die nächste Seite.

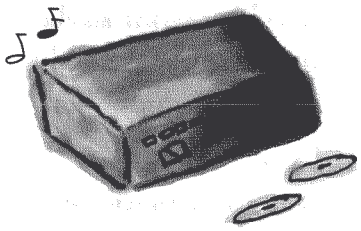
# Richtigen Sound erzeugen

Zu Beginn dieses Kapitels haben wir uns angesehen, in welcher Form MIDI-Daten die Anweisungen dafür enthalten, *was* gespielt werden soll (und *wie* es gespielt werden soll). Zudem haben wir festgestellt, dass die MIDI-Daten *nichts von dem Sound, den Sie hören, selbst erzeugen.* Damit Sound aus den Boxen kommt, müssen die MIDI-Daten in irgendein MIDI-Gerät eingespeist werden, das die MIDI-Anweisungen entgegennimmt und sie in Form von Sound ausgibt; dazu steuert es entweder ein Hardwareinstrument oder ein »virtuelles« Instrument (einen Software-Synthesizer) an. In diesem Buch verwenden wir nur Softwaregeräte. Und so funktioniert es in JavaSound:

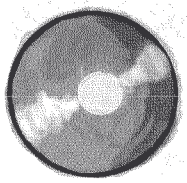
## VIER Dinge werden benötigt:

1. **Das Ding, das die Musik abspielt.**
2. **Die Musik, die gespielt wird ... ein Stück oder ein Song.**
3. **Der Teil der Sequenz, der die eigentliche Information enthält.**
4. **Die eigentlichen Informationen über die Musik: welche Töne gespielt werden, wie lang usw.**

**Sequencer** —spielt→ **Sequenz** —HAT EINen→ **Track** —enthält→ **Midi-Event**

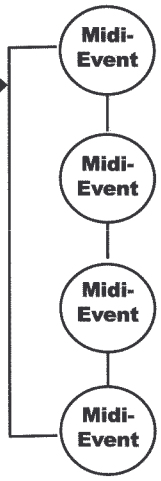


Der Sequencer ist das Ding, das das eigentliche Abspielen eines Stücks bewirkt. Stellen Sie es sich wie einen **Musik-CD-Player** vor.



Die Sequenz ist das Musikstück, das der Sequencer spielen wird. Für den Zweck dieses Buchs stellen Sie sich die Sequenz als eine Musik-CD vor – aber so, dass **die gesamte CD nur ein einziges Stück spielt.**

In diesem Buch brauchen wir nur einen einzigen Track, stellen Sie sich daher einfach eine Musik-CD mit nur genau einem Stück vor. Einem einzigen Track. Dieser Track ist es, in dem alle Daten für den Song (die MIDI-Informationen) stecken.




Ein MIDI-Ereignis oder -Event ist eine Nachricht, die der Sequencer verstehen kann. Ein Midi-Ereignis könnte (wenn es Deutsch spräche) beispielsweise sagen: »Zu diesem Zeitpunkt spiele das eingestrichene C, schlag es soundso fest an und halte es soundso lange.«

Ein Midi-Ereignis könnte auch so etwas sagen: »Wechsle das Instrument und nimm jetzt eine Querflöte.«

Für den Zweck dieses Buchs stellen Sie sich die Sequenz als eine CD mit nur einem Stück vor (d.h., sie hat nur einen einzigen Track). Die Informationen darüber, wie das Stück abzuspielen ist, stecken in dem Track, und der wiederum ist Teil der Sequenz.

## Und FÜNF Schritte sind erforderlich:

1. Holen Sie sich einen **Sequencer** und öffnen Sie ihn.

```
Sequencer player = MidiSystem.getSequencer();
player.open();
```

2. Erzeugen Sie eine neue **Sequenz**.

```
Sequence seq = new Sequence(timing,4);
```

3. Lassen Sie sich von der Sequenz einen neuen **Track** geben.

```
Track t = seq.createTrack();
```

4. Füllen Sie den Track mit **Midi-Ereignissen** (Midi-Events) und übergeben Sie die Sequenz an den Sequencer.

```
t.add(meinMidiEvent1);
player.setSequence(seq);
```

# Ihre allererste Sound-Player-Anwendung

Tippen Sie es ab und führen Sie es aus. Sie werden hören, wie irgendjemand einen einzelnen Ton auf dem Klavier spielt! (Okay, vielleicht nicht irgend*jemand*, aber irgend*etwas*.)

```
import javax.sound.midi.*;

public class MiniMiniMusikProg {

   public static void main(String[] args) {
      MiniMiniMusikProg mini = new MiniMiniMusikProg();
      mini.spielen();
   } // main schließen

   public void spielen() {
      try {

1        Sequencer player = MidiSystem.getSequencer();
         player.open();

2        Sequence seq = new Sequence(Sequence.PPQ, 4);

         Track track = seq.createTrack();

3

         ShortMessage a = new ShortMessage();
         a.setMessage(144, 1, 44, 100);
4        MidiEvent noteOn = new MidiEvent(a, 1);
         track.add(noteOn);

         ShortMessage b = new ShortMessage();
         b.setMessage(128, 1, 44, 100);
         MidiEvent noteOff = new MidiEvent(b, 16);
         track.add(noteOff);

         player.setSequence(seq);

         player.start();
         Thread.sleep(5000);
         player.close();

      } catch (Exception ex) {
         ex.printStackTrace();
      }
   } // play schließen
} // Klasse schließen
```

Vergessen Sie nicht, das Midi-Paket zu importieren.

Holen Sie sich einen Sequencer und öffnen Sie ihn (damit wir ihn benutzen können ... ein Sequencer wird nämlich nicht im offenen Zustand geliefert).

Machen Sie sich keine Gedanken über die Argumente für den Sequence-Konstruktor. Kopieren Sie einfach diese hier (betrachten Sie sie als Fertiggericht-Argumente).

Bitten Sie die Sequence um einen Track. Erinnern Sie sich: Der Track steckt in der Sequence, und die MIDI-Daten stecken im Track.

Stecken Sie ein paar Midi-Ereignisse in den Track. Dieser Abschnitt ist zum größten Teil Fertigcode. Sie müssen sich lediglich um die Argumente für die Methode setMessage() und die Argumente für den MidiEvent-Konstruktor kümmern. Wir sehen uns diese Argumente auf der nächsten Seite an.

Übergeben Sie die Sequence an den Sequencer (das entspricht dem Einlegen der CD in den CD-Player).

Starten Sie den Sequencer (das entspricht dem Drücken auf PLAY).

Damit der Player wieder geschlossen wird, nachdem er den Ton abgespielt hat (wie ein Thread funktioniert, werden wir in Kapitel 15 besprechen).

# Erzeugen eines MidiEvents (mit Musikstückdaten)

Ein MidiEvent ist eine Anweisung für einen Teil eines Musikstücks. Eine Folge von MidiEvents ist wie ein Notenblatt oder wie eine Notenrolle für ein automatisches Klavier. Die meisten MidiEvents, die uns interessieren, beschreiben ***etwas, was zu tun ist,*** und den ***Zeitpunkt, wann es zu tun ist.*** Der Zeitpunkt spielt eine wichtige Rolle, denn in der Musik dreht sich alles um Zeit. Dieser Ton folgt auf jenen Ton und so weiter. Und weil MidiEvents so detailgenau sind, müssen Sie angeben, in welchem Moment der Ton *anfangen* soll (ein NOTE-ON-Ereignis) und wann er *aufhören* soll (ein NOTE-OFF-Ereignis). Sie können sich sicher vorstellen, dass es nicht funktionieren würde, wenn »hör auf, den Ton G zu spielen« (ein NOTE-OFF-Ereignis) *vor* »fang an, den Ton G zu spielen« (einem NOTE-ON-Ereignis) käme.

Die MIDI-Anweisung steckt in der Realität in einem Message-Objekt; das Midi Event wiederum ist eine Kombination aus dieser Nachricht und dem Zeitpunkt, an dem die Nachricht »abgefeuert« werden soll. Mit anderen Worten, die Message könnte sagen: »Beginne das eingestrichene C zu spielen«, während das MidiEvent sagen würde: »Löse diese Nachricht bei Schlag 4 aus.«

Wir brauchen also immer eine Message und ein MidiEvent.

Die Message gibt an, *was* getan werden soll, und das MidiEvent gibt zudem an, *wann.*

**Ein MidiEvent gibt an, was zu tun ist und wann es zu tun ist.**

**Zu jeder Anweisung muss zudem das Timing angegeben werden.**

**Mit anderen Worten: bei welchem Schlag die Sache über die Bühne gehen soll.**

1. Machen Sie eine **Message**:

```
ShortMessage a = new ShortMessage();
```

2. Setzen Sie die **Anweisung** in die Message:

```
a.setMessage(144, 1, 44, 100);
```

Diese Message besagt: »Fang an, Ton 44 zu spielen.« (Die anderen Zahlen sehen wir uns auf der nächsten Seite noch an.)

3. Erzeugen Sie aus der Message ein neues **MidiEvent**:

```
MidiEvent noteOn = new MidiEvent(a, 1);
```

Die Anweisungen stecken in der Message, aber das MidiEvent fügt noch den Zeitpunkt hinzu, an dem die Anweisung ausgelöst werden sollte. Dieses MidiEvent gibt die Anweisung, die Message »a« beim ersten Schlag (Schlag 1) auszulösen.

4. Fügen Sie das MidiEvent zum **Track** hinzu:

```
track.add(noteOn);
```

Ein Track enthält alle MidiEvent-Objekte. Die Sequenz ordnet diese nach dem Zeitpunkt, an dem jedes Ereignis passieren soll, und der Sequencer gibt sie dann in genau dieser Reihenfolge wieder. Sie können eine Menge Events haben, die genau zum gleichen Zeitpunkt stattfinden. Beispielsweise möchten Sie ja vielleicht, dass zwei Töne gleichzeitig gespielt werden, oder sogar, dass verschiedene Instrumente unterschiedliche Töne zur gleichen Zeit spielen.

# MIDI-Message: das Herz eines MidiEvents

Eine MIDI-Message enthält den Teil des Events, der angibt, *was* zu tun ist. Die eigentliche Anweisung, die der Sequencer ausführen soll. Das erste Argument einer Anweisung ist immer der Typ der Nachricht. Die Werte, die Sie den anderen drei Argumenten übergeben, hängen vom Typ der Nachricht ab. So bedeutet beispielsweise eine Message vom Typ 144 »NOTE ON«. Aber um ein NOTE ON auszuführen, muss der Sequencer ein paar Dinge wissen. Stellen Sie sich vor, der Sequencer sagt: »Okay, ich spiele einen Ton, aber auf *welchem Kanal?* Oder anders ausgedrückt, soll ich einen Schlagzeugton spielen oder einen Klavierton? Und *welchen Ton?* Irgendein C? Dis? Und wenn wir schon mal dabei sind, mit *welcher Stärke* soll ich den Ton anschlagen?«

Um eine MIDI-Message zu erhalten, erzeugen Sie eine ShortMessage-Instanz; dann rufen Sie darauf setMessage() auf und übergeben die vier Argumente für die Message. Aber denken Sie daran – die Message gibt nur an, *was* zu tun ist; Sie müssen also die Nachricht auch noch in ein Event stecken, das zusätzlich die Information enthält, *wann* die Message »abgefeuert« werden soll.

## Anatomie einer Message

Das *erste* Argument für setMessage() repräsentiert immer den »Typ« der Message, während die *anderen* drei Argumente für unterschiedliche Dinge stehen (abhängig vom Message-Typ).

```
a.setMessage(144,  1,  44,  100);
```

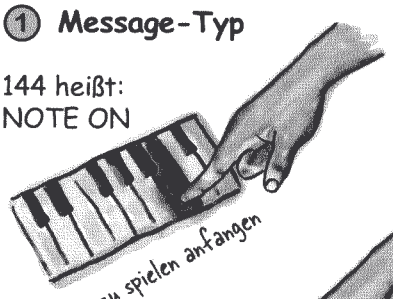
Message-Typ
Kanal
zu spielender Ton
Anschlagstärke

Die letzten drei Argumente variieren in Abhängigkeit vom Message-Typ. Dies ist eine NOTE-ON-Message, daher stehen die anderen drei Argumente für Dinge, die der Sequencer wissen muss, um einen Ton zu spielen.

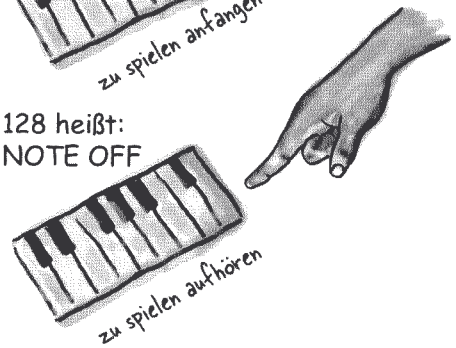
**Die *Message* gibt an, was zu tun ist, und das *MidiEvent*, wann dies zu tun ist.**

### ① Message-Typ

144 heißt: NOTE ON




128 heißt: NOTE OFF

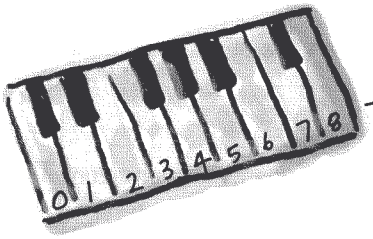



### ② Kanal

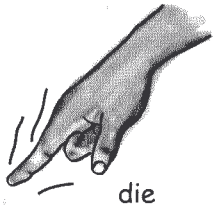
Stellen Sie sich einen Kanal wie einen Musiker in einer Band vor. Kanal 1 ist Musiker 1 (der Keyboard-Spieler), Kanal 9 ist der Schlagzeuger usw.

### ③ zu spielender Ton

Eine Zahl von 0 bis 127, von den tiefen zu den hohen Tönen.




### ④ Anschlagstärke



Wie fest haben Sie die Taste angeschlagen? 0 ist ein so weicher Anschlag, dass Sie wahrscheinlich gar nichts hören werden. 100 ist eine brauchbare Voreinstellung.

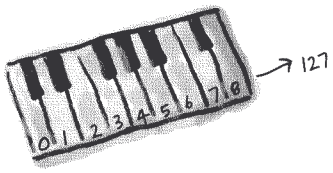
# Eine Message verändern

Nun wissen Sie, was hinter einer MIDI-Message steckt, und können anfangen zu experimentieren. Sie können verändern, welcher Ton gespielt und wie lange er gehalten wird, weitere Töne hinzufügen und sogar das Instrument wechseln.

## ① Den Ton verändern

Versuchen Sie es mit einer Zahl zwischen 0 und 127 in den note-on- und note-off-Messages.

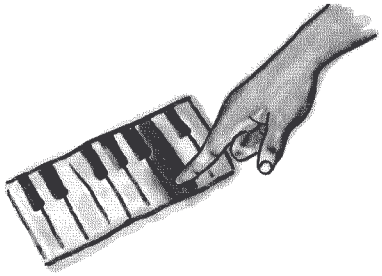
```
a.setMessage(144, 1, 20, 100);
```




## ② Die Dauer des Tons verändern

Ändern Sie das note-off-Event (nicht die *Message*) so, dass es bei einem früheren oder einem späteren Schlag stattfindet.

```
b.setMessage(128, 1, 44, 100);
MidiEvent noteOff = new MidiEvent(b, 3);
```



## ③ Das Instrument wechseln

Fügen Sie VOR der Message zum Abspielen des Tons eine neue Message ein, die das Instrument in Kanal 1 auf etwas anderes setzt als das standardmäßig vorgegebene Klavier. Die Instrumentwechsel-Message ist »192«, und das dritte Argument repräsentiert das eigentliche Instrument (versuchen Sie es mit einer Zahl zwischen 0 und 127).

```
first.setMessage(192, 1, 102, 0);
```

Instrumentwechsel-Message
auf Kanal 1 (Musiker 1)
zu Instrument 102 wechseln

## Version 2: Mit Kommandozeilen-Argumenten für das Experimentieren mit Sounds

Auch diese Version spielt noch immer lediglich einen einzelnen Ton, aber Sie können mit Hilfe von Argumenten, die Sie auf der Kommandozeile eingeben, das Instrument und die Tonhöhe verändern. Experimentieren Sie, indem Sie zwei int-Werte zwischen 0 und 127 übergeben. Der erste davon setzt das Instrument, der zweite den zu spielenden Ton.

```
import javax.sound.midi.*;

public class MiniMusikKommZeile {    // unsere zweite Version

  public static void main(String[] args) {
    MiniMusikKommZeile mini = new MiniMusikKommZeile();
    if (args.length < 2) {
      System.out.println("Vergessen Sie nicht Instrument- und Ton-Argument.");
    } else {
      int instrument = Integer.parseInt(args[0]);
      int ton = Integer.parseInt(args[1]);
      mini.spielen(instrument, ton);
    }
  } // main schließen

  public void spielen(int instrument, int ton) {

    try {

      Sequencer player = MidiSystem.getSequencer();
      player.open();
      Sequence seq = new Sequence(Sequence.PPQ, 4);
      Track track = seq.createTrack();

      ShortMessage first = new ShortMessage();
      first.setMessage(192, 1, instrument, 0);
      MidiEvent instrumentWechsel = new MidiEvent(first, 1);
      track.add(instrumentWechsel);

      ShortMessage a = new ShortMessage();
      a.setMessage(144, 1, ton, 100);
      MidiEvent noteOn = new MidiEvent(a, 1);
      track.add(noteOn);

      ShortMessage b = new ShortMessage();
      b.setMessage(128, 1, ton, 100);
      MidiEvent noteOff = new MidiEvent(b, 16);
      track.add(noteOff);
      player.setSequence(seq);
      player.start();
      Thread.sleep(5000);
      player.close();

} catch (Exception ex) {ex.printStackTrace();}
  } // spielen schließen
} // Klasse schließen
```

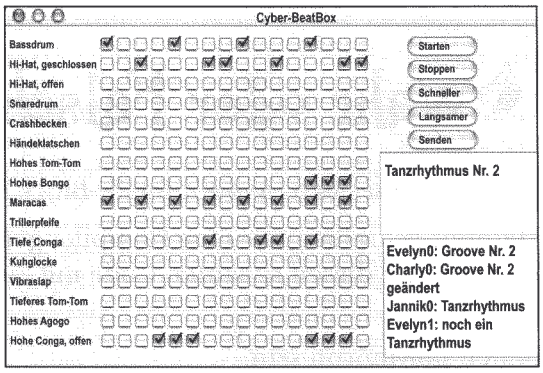
Führen Sie es mit zwei int-Argumenten zwischen 0 und 127 aus. Probieren Sie es für den Anfang mal mit diesen.

```
Datei Bearbeiten Fenster Hilfe NichtSoLaut
% java MiniMusikKommZeile 102 30
% java MiniMusikKommZeile 80 20
% java MiniMusikKommZeile 40 70
```

# Worauf wir mit unserer CodeKüche hinaus wollen:

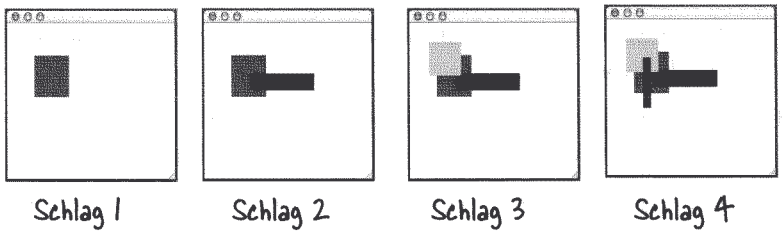
## Kapitel 15: Das Ziel

Wenn wir fertig sind, werden wir eine funktionierende BeatBox haben, die außerdem ein Schlagzeuger-Chat-Client ist. Dafür müssen wir vorher noch etwas über grafische Benutzeroberflächen (GUIs, einschließlich Event-Handling), E/A, Netzwerke und Threads lernen. Nur noch drei Kapitel (12, 13 und 14), dann sind wir so weit.



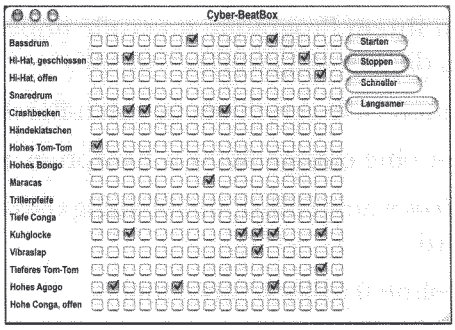

## Kapitel 12: Midi-Ereignisse

In dieser CodeKüche bauen wir ein kleines »Musikvideo« (na ja, diese Bezeichnung ist vielleicht ein wenig übertrieben), das zufällige Rechtecke zu den Schlägen der MIDI-Musik bildet. Wir lernen, wie man viele MidiEvents erstellt und abspielt (statt nur ein paar, wie in diesem Kapitel hier).



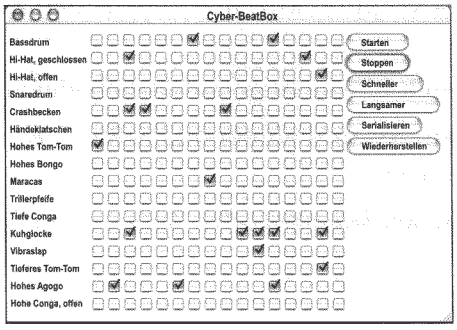

## Kapitel 13: Eine Standalone-BeatBox

Jetzt bauen wir die echte BeatBox, mit GUI und allem. Aber sie hat noch ein paar Einschränkungen – sobald Sie ein Pattern ändern, geht das vorherige verloren. Es gibt noch keine Möglichkeit zum Speichern und Wiederherstellen und auch keine Kommunikation mit dem Netzwerk. (Aber Sie können damit immerhin schon üben, immer bessere Drum-Patterns zu erstellen.)




## Kapitel 14: Speicherung und Wiederherstellung

Sie haben das perfekte Pattern erstellt und können es jetzt in einer Datei speichern. Und wieder laden, wenn Sie es erneut abspielen möchten. Damit sind wir bereit für die endgültige Version (Kapitel 15), wo wir das Pattern nicht in einer Datei speichern, sondern über ein Netz an den Chatserver senden werden.



In diesem Kapitel haben wir die wunderbare Welt der Exceptions erkundet. Entscheiden Sie, ob die folgenden Aussagen über Exceptions jeweils wahr oder falsch sind.

# Wahr oder Falsch?

1. Auf einen try-Block muss ein catch- *und* ein finally-Block folgen.
2. Wenn Sie eine Methode schreiben, die eine Compiler-geprüfte Exception auslösen könnte, *müssen* Sie diesen riskanten Code mit einem try/catch-Block umgeben.
3. Catch-Blöcke können polymorph sein.
4. Nur »Compiler-geprüfte« Exceptions können abgefangen werden.
5. Wenn Sie einen try/catch-Block definieren, muss nicht unbedingt ein finally-Block dabei sein.
6. Wenn Sie einen try-Block definieren, können Sie ihn mit einem catch- oder einem finally-Block oder mit beidem kombinieren.
7. Wenn Sie eine Methode schreiben, die deklariert, dass sie eine Compiler-geprüfte Exception auslösen kann, müssen Sie auch den Code, der die Exception auslöst, mit einem try/catch-Block umgeben.
8. Die main()-Methode in Ihrem Programm muss alle unbehandelten Exceptions behandeln, die ihr »zugeworfen« werden.
9. Auf *einen* try-Block können viele verschiedene catch-Blöcke folgen.
10. Eine Methode kann nur eine einzige Art von Exception auslösen.
11. Ein finally-Block wird *immer* ausgeführt, unabhängig davon, ob eine Exception ausgelöst wird.
12. Ein finally-Block kann ohne try-Block vorkommen.
13. Ein try-Block kann allein stehen, ohne einen catch- oder finally-Block.
14. Die Behandlung einer Exception wird manchmal auch als »Ausweichen« bezeichnet.
15. Die Reihenfolge der catch-Blöcke ist grundsätzlich gleichgültig.
16. Eine Methode mit einem try- und einem finally-Block *kann* die Exception deklarieren, *muss* es aber *nicht*.
17. Laufzeit-Exceptions müssen *behandelt* oder *deklariert* werden.

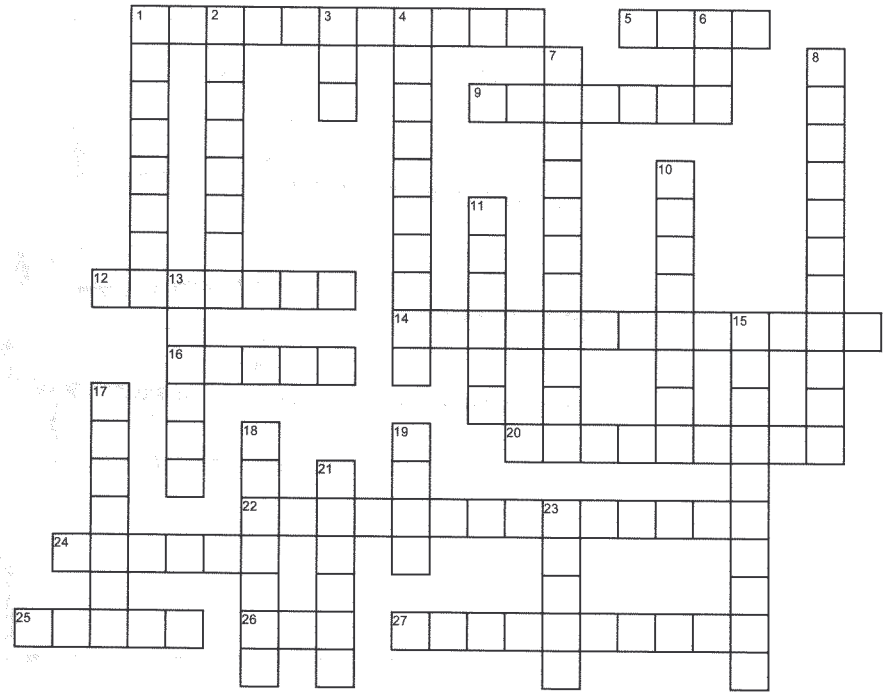
# Code-Magneten

Jemand hat ein funktionsfähiges Java-Programm in kleinen Schnipseln an den Kühlschrank geheftet. Können Sie alle Codeschnipsel wieder zu einem funktionierenden Java-Programm zusammensetzen, das die unten abgedruckte Ausgabe erzeugt? Ein paar geschweifte Klammern sind auf den Boden gefallen, und weil sie so klein sind, hat sie niemand aufgehoben – Sie dürfen also so viele davon einsetzen, wie Sie benötigen!

```
System.out.print("spa");
try {
System.out.print("au");
tuRiskantes(test);
System.out.println("n");
} finally {
System.out.print("nn");
class MeineEx extends Exception { }
public class ExTestlauf {
System.out.print("e");
if ("ja".equals(t)) {
System.out.print("loes");
throw new MeineEx();
} catch (MeineEx e) {
static void tuRiskantes(String t) throws MeineEx {
System.out.print("s");
public static void main(String [] args) {
String test = args[0];
```

```
Datei Bearbeiten Fenster Hilfe TotaleErschöpfung
% java ExTestlauf ja
ausloesen

% java ExTestlauf nein
ausspannen
```

# JavaKreuzworträtsel 7.0



Sie wissen ja, wie es geht!

## Waagerecht

1. Code-Rezept.
5. Überwiegend statische API-Klasse.
9. Zu guter Letzt.
12. Nach einer Vorlage erzeugt.
14. Mach noch einen, Sam.
16. Eine schlechte Nachricht loslassen.
20. Nicht ausweichen.
22. Eins aus Javas '49'.
24. Besser nicht den Kindern zeigen.
25. Abfangen.
26. Erzeugung in Gang setzen.
27. Nicht behandeln.

## Senkrecht

1. Starten.
2. Von Javac rechtzeitig bemerkt.
3. Häufiger primitiver Typ.
4. Der Familienstammbaum.
6. Risiko eingehen.
7. Kann alles – und noch mehr!
8. Ausweichmanöver ankündigen.
10. Wert verleihen.
11. Vorlage für Instanzen.
13. Ändernde Methode.
15. Problemobjekte.
17. Picasso eher nicht.
18. Verhaltensunabhängig.
19. Klassenhierarchie.
21. Eine aufsässige Methode tut es.
23. Automatisch Besitz erlangen.

## Zusatzhinweise:

**Waagerecht:**
5. Zahlenkram.
20. Riskantes mit try/catch umgeben.
22. Ruft ein Problem hervor.
24. Juristisches.
27. Im Straßenverkehr manchmal ratsam.

**Senkrecht:**
1. Eine Methode an die Arbeit kriegen.
7. Ein Java-Kind.
8. Müssen Sie beim Zoll.
13. Manche bellen.
15. Ausnahmen.
17. Nichts Abstraktes.
19. Holzlieferant.
23. Das Familienvermögen _____.

# Lösungen zu den Übungen

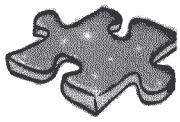
## Wahr oder Falsch

1. Falsch; einer von beiden oder beide.
2. Falsch; man kann die Exception auch deklarieren.
3. Wahr.
4. Falsch; eine Laufzeit-Exception kann abgefangen werden.
5. Wahr.
6. Wahr; beide sind erlaubt.
7. Falsch; die Deklaration reicht.
8. Falsch; aber wenn sie es nicht tut, kann die JVM geschlossen werden.
9. Wahr.
10. Falsch.
11. Wahr. Das wird häufig zum »Aufräumen« nach unvollständig abgeschlossenen Tasks verwendet.
12. Falsch.
13. Falsch.
14. Falsch; Ausweichen ist gleichbedeutend mit Deklarieren.
15. Falsch; die weitesten Exceptions müssen von den letzten catch-Blöcken abgefangen werden.
16. Falsch; wenn Sie keinen catch-Block haben, *müssen* Sie sie deklarieren.
17. Falsch.

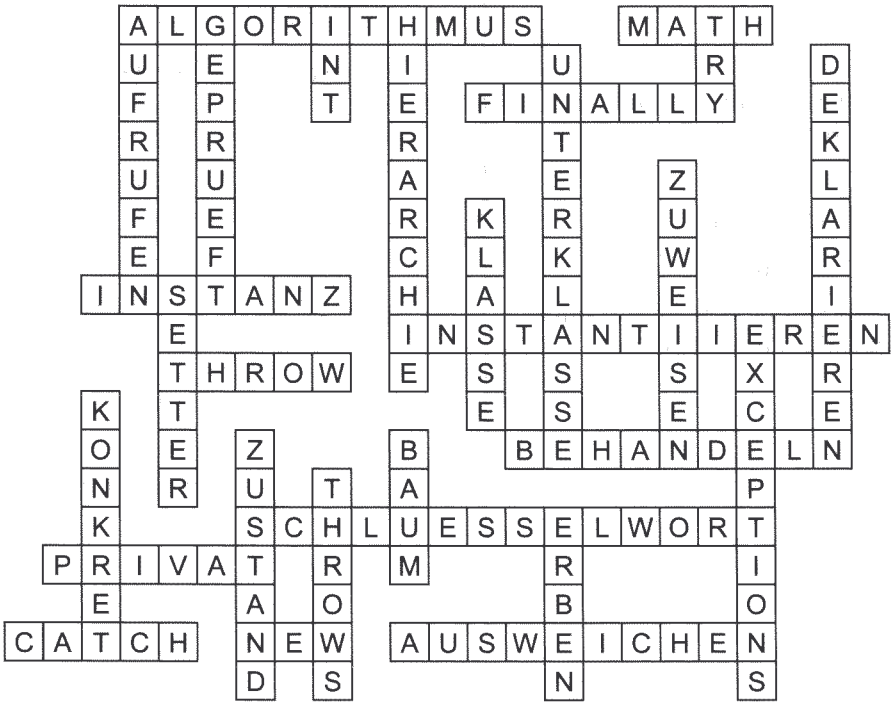
## Code-Magneten

```
class MeineEx extends Exception { }

public class ExTestlauf {

  public static void main(String [] args) {
    String test = args[0];
    try {

      System.out.print("au");

      tuRiskantes(test);

      System.out.print("nn");

    } catch (MeineEx e) {

      System.out.print("loes");

    } finally {

      System.out.print("e");
    }

    System.out.println("n");
  }

  static void tuRiskantes(String t) throws MeineEx {
    System.out.print("s");

    if ("ja".equals(t)) {
      throw new MeineEx();
    }

    System.out.print("spa");
  }
}
```

```
Datei Bearbeiten Fenster Hilfe Relaxen
% java ExTestlauf ja
ausloesen

% java ExTestlauf nein
ausspannen
```



# Lösung zum JavaKreuzworträtsel



# Innen hui, außen GUI

**Sehen Sie den Tatsachen ins Auge: Um GUIs (grafische Benutzeroberflächen) kommen Sie nicht herum.** Wenn Sie Anwendungen schreiben, die von anderen benutzt werden sollen, *brauchen* Sie eine grafische Schnittstelle. Wenn Sie Programme nur für sich selbst schreiben, *wollen* Sie eine grafische Schnittstelle. Selbst wenn Sie davon ausgehen, dass Sie den Rest Ihres Lebens serverseitigen Code – mit einer Webseite als Client-Benutzerschnittstelle – schreiben, müssen Sie früher oder später Tools schreiben, und dafür hätten Sie dann ganz sicher gern eine grafische Schnittstelle. Klar, Kommandozeilen-Anwendungen sind so schön »retro« – aber nicht im positiven Sinn. Sie sind bei aller Leistungsfähigkeit einfach nicht so schick und benutzerfreundlich. Wir werden zwei Kapitel mit der Programmierung von GUIs zubringen und dabei grundlegende Features der Sprache Java kennen lernen, z.B. **Event-Handling** und **innere Klassen.** In diesem Kapitel setzen wir einen Button auf den Bildschirm und lassen ihn reagieren, wenn man ihn anklickt. Außerdem zeichnen wir etwas auf den Bildschirm, zeigen ein JPEG-Bild an und versuchen uns sogar an einer kleinen Animation.

## Alles beginnt mit einem Fenster

Ein JFrame ist ein Objekt, das ein Fenster auf dem Bildschirm repräsentiert. Dort bringen Sie all Ihren Oberflächenkram unter, wie z.B. Buttons, Kontrollkästchen, Textbereiche usw. Es kann auch eine waschechte Menüleiste mit Menüpunkten enthalten. Und es gehören all die kleinen Fenster-Icons dazu, die Sie (auf welcher Plattform auch immer) zum Minimieren, Maximieren und Schließen des Fensters brauchen.

Wie der JFrame genau aussieht, hängt von Ihrer Plattform ab. Dies ist ein JFrame auf Mac OS X:

ein JFrame mit einer Menüleiste und zwei »Widgets« (einem Button und einem Radio-Button)

**»Wenn ich noch eine einzige Kommandozeilen-Anwendung zu sehen kriege, werden Sie gefeuert!«**

## Widgets in ein Fenster setzen

In einen JFrame – eine Art »Rahmen« – können Sie andere Dinge (»Widgets«, was so viel wie »kleine Geräte« bedeutet) hineinsetzen, indem Sie diese mit add() zum JFrame hinzufügen. Es gibt eine Unmenge an Swing-Komponenten, die man hinzufügen kann; schauen Sie im Paket javax.swing nach. Zu den gebräuchlichsten gehören JButton, JRadioButton, JCheckBox, JLabel, JList, JScrollPane, JSlider, JTextArea, JTextField und JTable. Die meisten sind wirklich einfach zu verwenden, aber manche (z.B. JTable) können ein bisschen komplizierter sein.

### Eine GUI zu erstellen ist leicht:

1. Erzeugen Sie einen »Rahmen« (einen JFrame):
```
JFrame frame = new JFrame();
```

2. Erzeugen Sie ein Widget (Button, Textfeld usw.):
```
JButton button = new JButton("klick mich");
```

3. Fügen Sie das Widget zum Frame hinzu:
```
frame.getContentPane().add(button);
```

Sie fügen Dinge nicht direkt zum Frame hinzu. Stellen Sie sich den Frame als den Rahmen um das Fenster vor, so dass Sie die Dinge zur Fensterscheibe (engl. pane) hinzufügen. (So mussten Sie zumindest vor Java 5 vorgehen – aber in Java 5 dürften Sie jetzt sogar direkt frame.add(button) schreiben! In diesem Buch benutzen wir jedoch noch den Weg über getContentPane().)

4. Zeigen Sie ihn an (geben Sie ihm eine Größe und machen Sie ihn sichtbar):
```
frame.setSize(300,300);
frame.setVisible(true);
```

# Ihre erste GUI: ein Button in einem Frame

```
import javax.swing.*;

public class EinfacheGui1 {
  public static void main (String[] args) {

    JFrame frame = new JFrame();
    JButton button = new JButton("klick mich");

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    frame.getContentPane().add(button);

    frame.setSize(300,300);

    frame.setVisible(true);
  }
}
```

Vergessen Sie nicht, dieses Swing-Paket zu importieren.

Erzeugen Sie einen Frame und einen Button. (Sie können dem Button-Konstruktor den Text übergeben, der auf dem Button stehen soll.)

Diese Zeile sorgt dafür, dass beim Schließen des Fensters auch das Programm beendet wird (wenn Sie die Zeile weglassen, bleibt es einfach für ewig und alle Zeiten auf dem Schirm hocken).

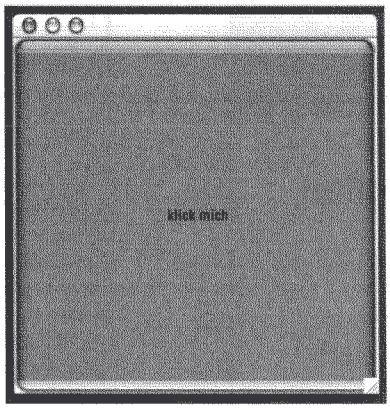
Fügen Sie den Button zur ContentPane – der »Fensterscheibe« im Frame – hinzu.

Geben Sie dem Frame eine Größe (in Pixeln).

Und zum Schluss machen Sie ihn sichtbar! (Wenn Sie diesen Schritt vergessen, werden Sie beim Ausführen dieses Codes nichts zu sehen bekommen.)

**Sehen wir uns an, was beim Ausführen passiert:**

```
%java EinfacheGui1
```




## Huch! Das ist ja ein echt großer Button.

Der Button nimmt den gesamten verfügbaren Platz im Frame ein. Wir lernen später noch, wie man steuert, wo der Button innerhalb des Frames steht (und wie groß er ist).

## Aber es passiert nichts, wenn ich draufklicke ...

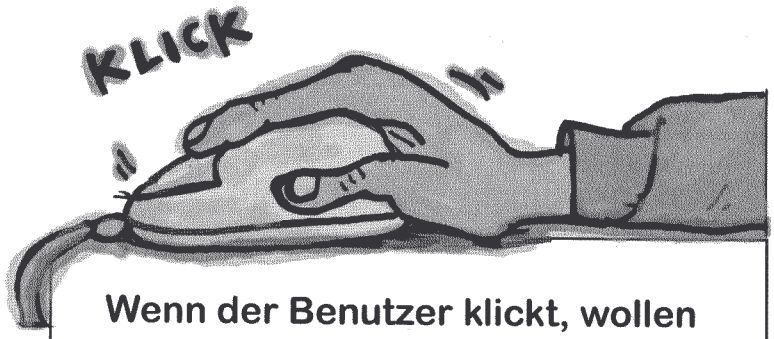
Das stimmt nicht ganz. Wenn Sie den Button drücken, zeigt er dieses typische »runtergedrückte« Aussehen (was vom »Look-and-Feel« auf der jeweiligen Plattform abhängt; aber *irgendwie* lässt der Button es auf jeden Fall erkennen, dass er gedrückt ist).

Die eigentliche Frage lautet: »Wie bringe ich den Button dazu, etwas Bestimmtes zu tun, wenn der Benutzer ihn anklickt?«

### Zwei Dinge brauchen wir dazu:

1. Eine **Methode**, die aufgerufen wird, wenn der Benutzer klickt (damit das passiert, was Sie als Resultat des Button-Klicks vorgesehen haben).

2. Eine Möglichkeit, um zu **erfahren**, wann die Methode aufgerufen werden soll. Mit anderen Worten: eine Möglichkeit, um zu erfahren, wann der Benutzer den Button anklickt!




**Wenn der Benutzer klickt, wollen wir das erfahren.**

**Wir interessieren uns für das »Benutzer-führt-eine-Aktion-auf-einem-Button-aus«-Ereignis.**

### Es gibt keine Dummen Fragen

**F: Sieht ein Button unter Windows wie ein Windows-Button aus?**

A: Wenn Sie das wollen, ja. Sie können unter ein paar »Look-and-Feel«-Klassen in der Kernbibliothek wählen, die steuern, wie eine Oberfläche aussieht. In den meisten Fällen können Sie zwischen mindestens zwei verschiedenen Möglichkeiten wählen: dem Java-Standard-Look-and-Feel (auch als ***Metal*** bezeichnet) und Ihrem plattformtypischen Look-and-Feel. Die Mac OS X-Bildschirmfotos in diesem Buch zeigen entweder das OS X-***Aqua***-Look-and-Feel oder das ***Metal***-Look-and-Feel.

**F: Kann ich ein Programm *immer* wie Aqua aussehen lassen? Auch wenn es unter Windows läuft?**

A: Nein. Es stehen nicht für jede Plattform alle Look-and-Feels zur Verfügung. Wenn Sie auf Nummer sicher gehen wollen, können Sie das Look-and-Feel explizit auf Metal setzen, dann wissen Sie genau, was Sie bekommen – egal, wo die Anwendung läuft. Oder Sie geben gar kein Look-and-Feel an und akzeptieren das jeweils voreingestellte.

**F: Ich habe gehört, Swing wäre eine lahme Ente und würde von niemandem verwendet.**

A: Das galt früher einmal, heute jedoch nicht mehr. Bei leistungsschwachen Rechnern bekommen Sie es vielleicht noch zu spüren. Aber auf neueren Computern und seit der Java-Version 1.3 und höher werden Sie den Unterschied zwischen einer Swing-Oberfläche und einer nativen Oberfläche vielleicht nicht einmal bemerken. Swing wird heute sehr verbreitet eingesetzt, und zwar in den unterschiedlichsten Anwendungen.

# Wie man an ein Benutzerereignis kommt

Stellen Sie sich vor, Sie möchten den Text *klick mich* auf dem Button in *ich wurde geklickt!* ändern, wenn der Benutzer den Button anklickt. Dazu können wir erst einmal eine Methode schreiben, die den Text des Buttons verändert (ein rascher Blick in die API zeigt Ihnen die entsprechende Methode von Button):

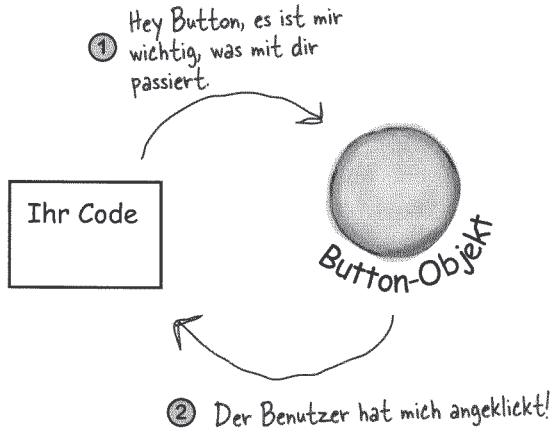
```
public void ändern() {
  button.setText("ich wurde geklickt!");
}
```

Aber was *jetzt?* Wie *erfahren* wir, wann diese Methode ausgeführt werden soll? ***Wie erfahren wir, wann der Button angeklickt wird?***

In Java bezeichnet man das Verfahren, wie man an ein Benutzerereignis – ein »Event« – gelangt und wie man es behandelt, als *Event-Handling*. Es gibt viele verschiedene Arten von Events in Java, die meisten davon hängen allerdings mit GUI-Benutzeraktionen zusammen. Wenn der Benutzer auf einen Button klickt, ist das ein Event. Ein Event, das besagt: »Der Benutzer will, dass die Aktion für diesen Button stattfindet.« Wenn es ein »Tempo verlangsamen«-Button ist, möchte der Benutzer, dass die Tempo-verlangsamen-Aktion ausgeführt wird. Wenn es ein Senden-Button ist oder ein Chat-Client, will der Benutzer, dass die Sende-meine-Nachricht-Aktion ausgeführt wird. Das einfachste Event ist also, wenn der Benutzer auf den Button klickt und damit kundtut: »Ich möchte, dass eine Aktion stattfindet.«

Bei Buttons interessieren Sie sich normalerweise nicht für irgendwelche Zwischenereignisse wie z.B. das Herunterdrücken oder das Wiederloslassen. Was Sie dem Button sagen möchten, ist Folgendes: »Es ist mir gleich, wie der Benutzer mit dem Button herumspielt, wie lange er den Mauszeiger darüber stehen lässt, wie oft er es sich anders überlegt und den Mauszeiger wieder herunterbewegt, bevor er loslässt, usw. ***Sag es mir einfach, wenn der Benutzer es wirklich ernst meint!*** Mit anderen Worten, ruf mich nur dann auf, wenn der Benutzer in einer Weise klickt, die deutlich macht: Er will, dass der verdammte Button das tut, was auf ihm geschrieben steht!«

**Erst einmal muss der Button wissen, dass er uns nicht egal ist.**




**Zweitens braucht der Button eine Möglichkeit, uns zu benachrichtigen, wenn ein Button-geklickt-Event eintritt.**

1) Wie könnten Sie einem Button-Objekt mitteilen, dass Ihnen seine Events wichtig sind? Dass Sie interessiert darauf lauschen?

2) Wie wird der Button Sie benachrichtigen? Nehmen Sie an, es gibt keine Möglichkeit, wie Sie dem Button den Namen Ihrer ganz speziellen Methode (ändern()) mitteilen können. Wodurch können wir dem Button sonst noch klarmachen, dass wir eine spezielle Methode haben, die er aufrufen kann, wenn das Event stattfindet?

**Wenn Sie sich für die Ereignisse des Buttons interessieren, implementieren Sie ein Interface, das ihm mitteilt: »Ich *lausche* auf deine Events.«**

Ein **Listener-Interface** ist die Brücke zwischen dem **Listener** (dem »Lauscher«, also Ihnen) und der **Event-Quelle** (dem Button).

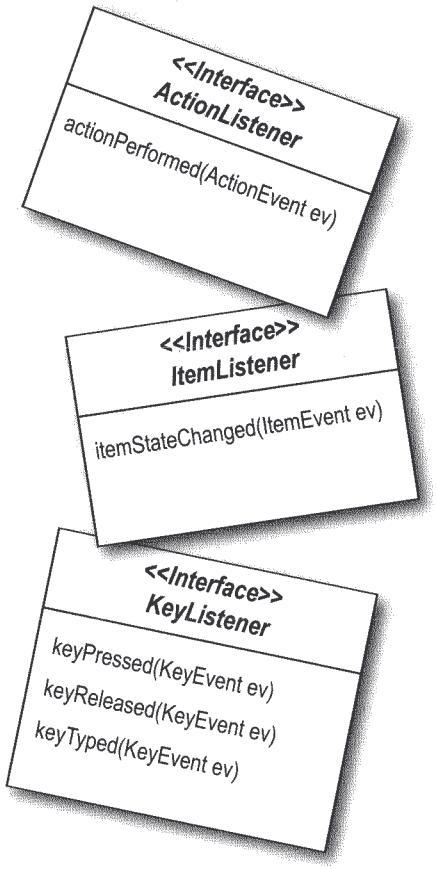
Die Swing-GUI-Komponenten sind Event-Quellen. Im Java-Zusammenhang ist eine Event-Quelle ein Objekt, das Benutzeraktionen (den Mausklick, das Drücken einer Taste, das Schließen eines Fensters) in Events umwandelt. Und wie nahezu alles andere in Java wird auch ein Event als Objekt repräsentiert. Ein Objekt irgendeiner Event-Klasse. Wenn Sie das Paket java.awt.event in der API durchsehen, finden Sie einen ganzen Schwung von Event-Klassen (leicht daran zu erkennen, dass sie alle ***Event*** im Namen tragen). Da gibt es z.B. MouseEvent, KeyEvent, WindowEvent, ActionEvent und diverse andere.

Eine Event-***Quelle*** (wie z.B. ein Button) erzeugt ein ***Event-Objekt***, wenn der Benutzer etwas tut, was von Belang ist (wie z.B. den Button *anklicken*). Der größte Teil Ihres selbst geschriebenen Codes (und der gesamte Code in diesem Buch) wird keine Events *erzeugen*, sondern sie *empfangen*. Anders ausgedrückt: Den größten Teil Ihrer Zeit werden Sie als Event-*Listener* verbringen, nicht als Event-*Quelle*.

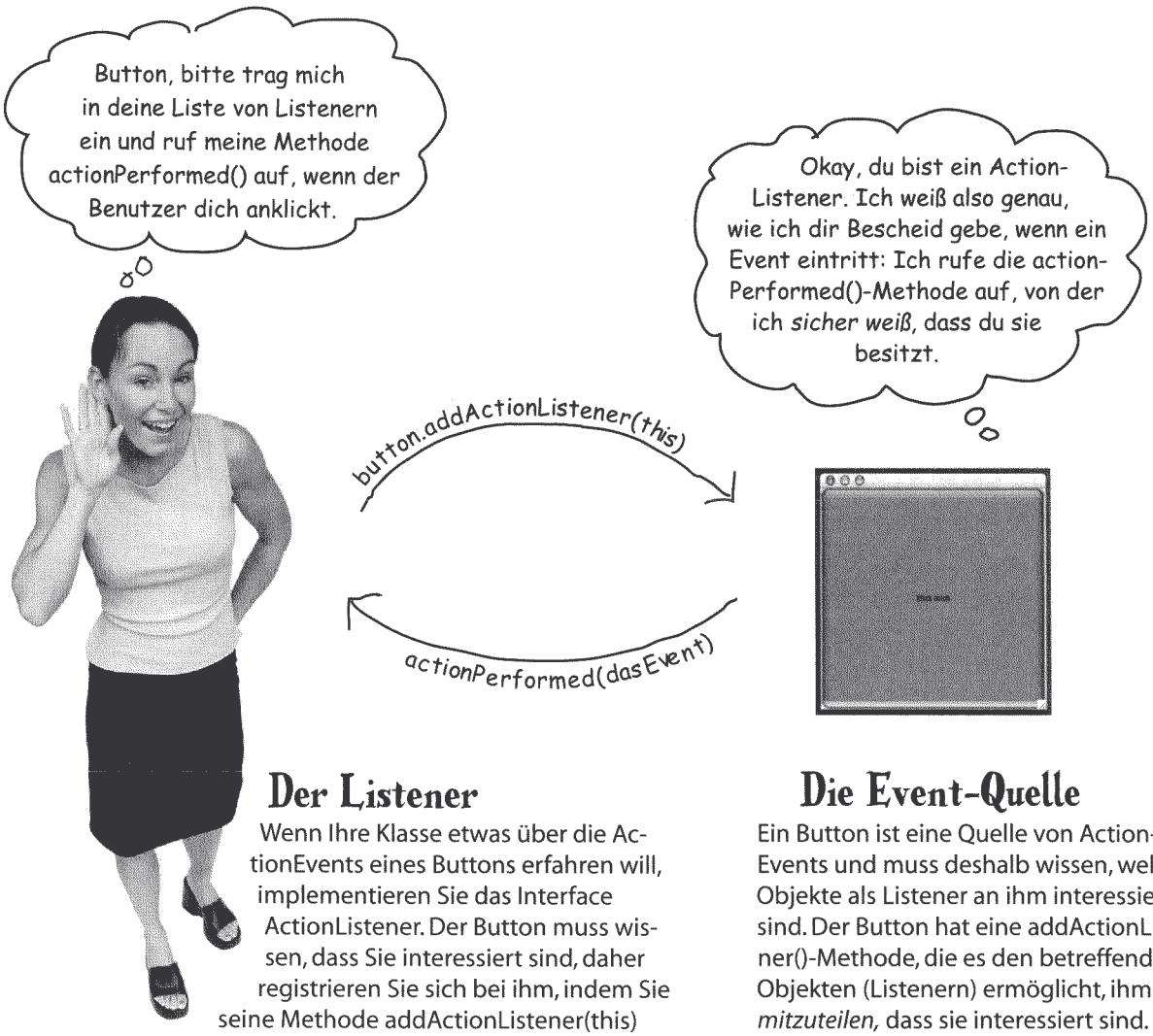
Zu jedem Event-Typ gehört ein passendes Listener-Interface. Wenn Sie MouseEvents haben wollen, implementieren Sie das MouseListener-Interface. Sie wollen WindowEvents? Implementieren Sie WindowListener. Sie verstehen das Prinzip? Und erinnern Sie sich an die Regeln zu den Interfaces – um ein Interface zu implementieren, *deklarieren* Sie, dass Sie es implementieren (class Hund implements Haustier), was bedeutet, dass Sie selbst eine *Implementierungsmethode* für jede der Methoden aus dem Interface schreiben müssen.

Manche Interfaces enthalten mehr als eine Methode, weil das Event selbst in unterschiedlichen Geschmacksrichtungen zu haben ist. Wenn Sie beispielsweise MouseListener implementieren, können Sie mousePressed-, mouseReleased-, mouseMoved-Events usw. erhalten. Zu jedem dieser Mausereignisse gehört eine eigene Methode im Interface, auch wenn sie alle ein MouseEvent entgegennehmen. Wenn Sie MouseListener implementieren, wird die Methode mousePressed() aufgerufen, wenn der Benutzer – richtig geraten! – die Maustaste drückt. Und wenn der Benutzer loslässt, wird die Methode mouseReleased() aufgerufen. Bei Mausereignissen gibt es also nur ein einziges Event-*Objekt* – MouseEvent –, aber mehrere verschiedene Event-*Methoden*, die die unterschiedlichen *Arten* von Mausereignissen repräsentieren.

Wenn Sie ein Listener-Interface implementieren, geben Sie damit dem Button eine Möglichkeit, Sie zu benachrichtigen. In dem Interface wird die Methode für die Benachrichtigung deklariert.



## Wie sich der Listener und die Quelle verständigen:




### Der Listener

Wenn Ihre Klasse etwas über die ActionEvents eines Buttons erfahren will, implementieren Sie das Interface ActionListener. Der Button muss wissen, dass Sie interessiert sind, daher registrieren Sie sich bei ihm, indem Sie seine Methode addActionListener(this) aufrufen und ihr eine ActionListener-Referenz übergeben (in diesem Fall sind *Sie* der ActionListener, also übergeben Sie *this*). Der Button braucht eine Möglichkeit, Sie zu benachrichtigen, wenn das Event eintritt, daher ruft er die Methode aus dem Listener-Interface auf. Als ActionListener *müssen* Sie die einzige Methode aus diesem Interface – actionPerformed() – implementieren. Das garantiert der Compiler.

### Die Event-Quelle

Ein Button ist eine Quelle von ActionEvents und muss deshalb wissen, welche Objekte als Listener an ihm interessiert sind. Der Button hat eine addActionListener()-Methode, die es den betreffenden Objekten (Listenern) ermöglicht, ihm *mitzuteilen,* dass sie interessiert sind.

Wird die Methode addActionListener() des Buttons ausgeführt (weil ein potenzieller Listener sie aufgerufen hat), nimmt der Button den Parameter (eine Referenz auf das Listener-Objekt) und speichert ihn in einer Liste. Wenn der Benutzer den Button anklickt, wird das Event vom Button abgesetzt (»abgefeuert«), indem dieser auf jedem Listener in seiner Liste die Methode actionPerformed() aufruft.

## Wie man an das ActionEvent eines Buttons kommt

1. Implementieren Sie das Interface ActionListener.
2. Registrieren Sie sich beim Button (teilen Sie ihm mit, dass Sie auf Events lauschen wollen).
3. Definieren Sie die Event-Handling-Methode (implementieren Sie die Methode actionPerformed() aus dem Interface ActionListener).

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

Eine neue Importanweisung für das Paket, zu dem ActionListener und ActionEvent gehören.

```
public class EinfacheGui1B implements ActionListener {
  JButton button;

  public static void main (String[] args) {
    EinfacheGui1B gui = new EinfacheGui1B();
    gui.los();
  }

  public void los() {
    JFrame frame = new JFrame();
    button = new JButton("klick mich");

    button.addActionListener(this);

    frame.getContentPane().add(button);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300,300);
    frame.setVisible(true);
  }

  public void actionPerformed(ActionEvent event) {
    button.setText("Ich wurde geklickt!");
  }
}
```

1 Implementieren Sie das Interface. Das bedeutet: »Eine Instanz von EinfacheGui1B IST-EIN ActionListener.«

(Der Button schickt Events nur an registrierte Implementierer von ActionListener.)

2 Registrieren Sie sich beim Button. Damit sagen Sie zum Button: »Nimm mich in deine Listener-Liste auf.« Das von Ihnen übergebene Argument MUSS ein Objekt einer Klasse sein, die ActionListener implementiert!!

3 Implementieren Sie die Methode actionPerformed() aus dem Interface ActionListener. Dies ist die eigentliche Event-Handling-Methode!

Der Button ruft diese Methode auf und lässt Sie damit wissen, dass ein Event eingetreten ist. Als Argument sendet er Ihnen ein ActionEvent-Objekt, aber das benötigen wir nicht. Es genügt uns zu wissen, dass das Event eingetreten ist.

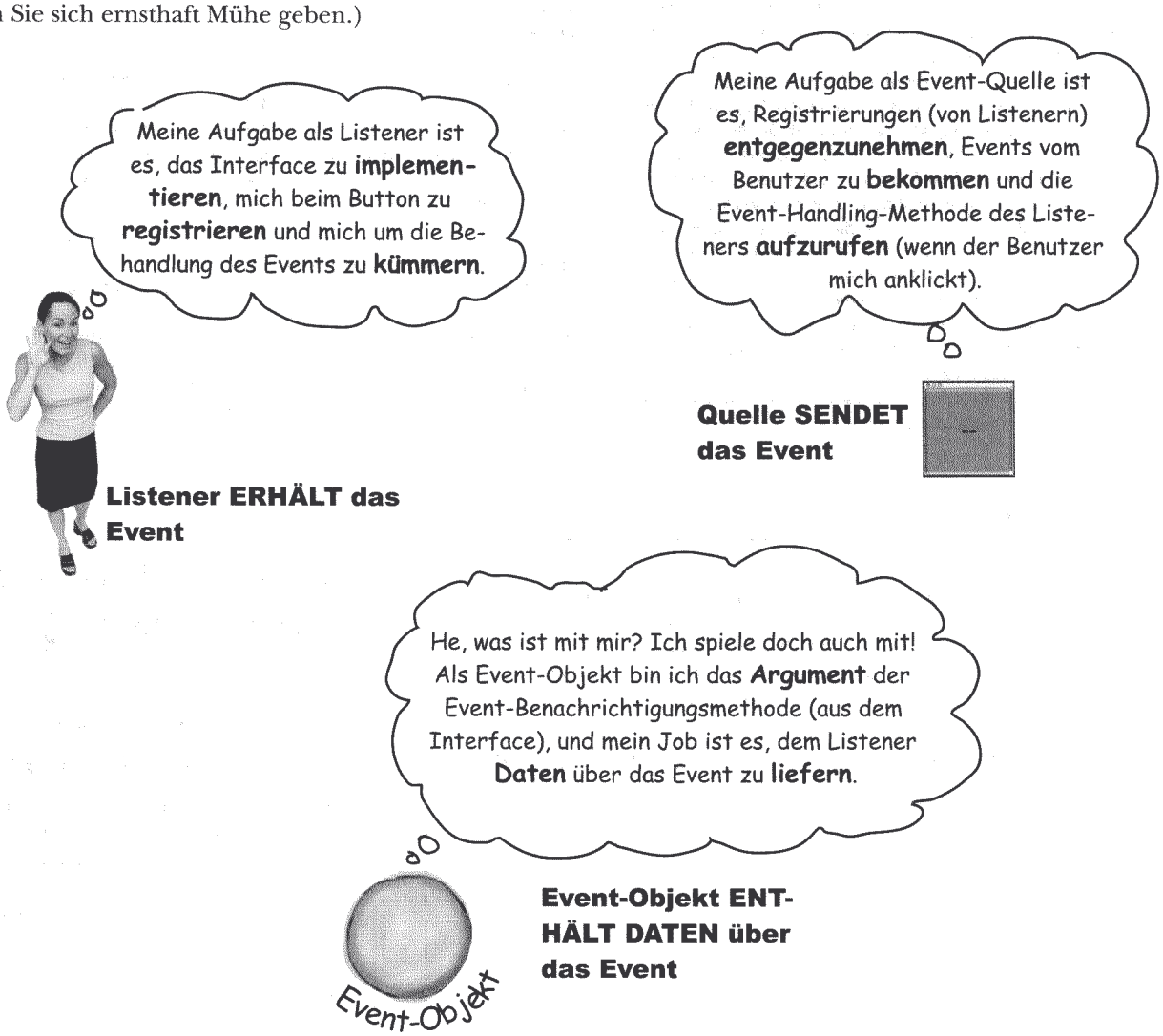
# Listener, Quellen und Events

Im größten Teil Ihrer Java-Karriere in diesem Universum werden nicht *Sie* die *Quelle* von Events sein.

(Wie sehr Sie sich selbst auch als das Zentrum Ihres sozialen Universums sehen mögen.)

Gewöhnen Sie sich daran. ***Ihr Job ist es, als Listener gut hinzuhören.***

(Was Ihre sozialen Beziehungen in der Tat verbessern *kann*, wenn Sie sich ernsthaft Mühe geben.)



## Es gibt keine Dummen Fragen

**F: Warum kann ich keine Quelle von Events sein?**

A: Doch, *können* Sie. Wir haben nur gesagt, dass Sie *fast immer* der Empfänger des Events sein werden und nicht sein Ursprung (zumindest *am Beginn* Ihrer glanzvollen Java-Karriere). Die meisten Events, die Sie interessieren könnten, werden von Klassen der Java-API »abgefeuert«, und Sie müssen nichts weiter tun, als Listener für diese Events zu sein. Aber vielleicht entwerfen Sie ja auch ein Programm, bei dem Sie ein selbst definiertes Event brauchen – z.B. ein AktienMarktEvent, das ausgelöst wird, wenn Ihre Aktienmarktbeobachtungsanwendung etwas entdeckt, das sie als wichtig einstuft. In diesem Fall würden Sie das AktienBeobachter-Objekt zu einer Event-Quelle machen und das Gleiche tun, was auch ein Button (oder irgendeine andere Quelle) tut: ein Listener-Interface für Ihr selbst definiertes Event schreiben und eine Registrierungsmethode zur Verfügung stellen (addAktienListener()), und wenn diese von jemandem aufgerufen wird, wird der Aufrufer (ein Listener) in die Listener-Liste eingetragen. Wenn dann ein Aktienereignis eintritt, instantiieren Sie ein AktienEvent-Objekt (eine weitere selbst geschriebene Klasse) und senden es an die Listener auf Ihrer Liste, indem Sie deren Methode aktienVerändert(AktienEvent ev) aufrufen. Und vergessen Sie nicht, dass es zu jedem Event-Typ ein *passendes Listener-Interface* geben muss (d.h., Sie schreiben ein AktienListener-Interface mit einer Methode aktienVerändert()).

**F: Die Bedeutung des Event-Objekts, das den Methoden zur Event-Benachrichtigung übergeben wird, ist mir noch nicht klar. Wenn jemand meine mousePressed()-Methode aufruft, brauche ich doch sonst keine weiteren Informationen, oder?**

A: Die meiste Zeit und bei den meisten Entwürfen brauchen Sie das Event-Objekt nicht. Es ist nichts weiter als ein kleiner Datenträger, mit dem sich zusätzliche Informationen über das Ereignis übermitteln lassen. Aber manchmal möchten Sie vielleicht bestimmte Einzelheiten über das Event wissen. Wenn beispielsweise Ihre mousePressed()-Methode aufgerufen wird, wissen Sie, dass Ihre Maustaste heruntergedrückt wurde. Aber wenn Sie nun wissen möchten, wo genau diese gedrückt wurde? Anders ausgedrückt, wenn Sie wissen wollen, bei welchen x- und y-Bildschirmkoordinaten die Maustaste gedrückt wurde?

Oder: Manchmal möchten Sie den *gleichen* Listener bei *mehreren* Objekten registrieren. Ein Bildschirmtaschenrechner beispielsweise hat zehn Zahlentasten, und da diese alle das Gleiche tun, möchten Sie nicht für jede einzelne Taste einen separaten Listener machen. Stattdessen könnten Sie einen einzigen Listener bei allen zehn Tasten registrieren, und wenn ein Event bei Ihnen eingeht (weil Ihre Event-Benachrichtigungsmethode aufgerufen wird), können Sie eine Methode auf dem Event-Objekt aufrufen, um herauszufinden, *wer* denn nun die Event-Quelle war bzw. *welche Taste dieses Event geschickt hat.*

## Spitzen Sie Ihren Bleistift

Jedes dieser Widgets (Benutzeroberflächenobjekte) ist die Quelle für ein oder mehrere Events. Ordnen Sie die Widgets den von Ihnen aufgerufenen Methoden zu. Manche der Widgets können als Quelle für mehr als ein Event fungieren, und manche der Events können von mehr als einem der Widgets erzeugt werden.

| Widgets | Event-Methoden |
|---|---|
| Checkbox | windowClosing() |
| Textfeld | actionPerformed() |
| scrollbare Liste | itemStateChanged() |
| Button | mousePressed() |
| Dialogfeld | keyTyped() |
| Radio-Button | mouseExited() |
| Menüpunkt (engl. menu item) | focusGained() |

## Woher WEISS man, ob ein Objekt eine Event-Quelle ist?

**Sehen Sie in der API nach.**

**OK. Und wonach suche ich?**

**Nach einer Methode, die mit »add« beginnt, mit »Listener« aufhört und ein Listener-Interface-Argument übergeben bekommt. Wenn Sie so etwas sehen:**

`addKeyListener(KeyListener k)`

**wissen Sie, dass die Klasse mit dieser Methode eine Quelle für KeyEvents ist. Es gibt eine Namenskonvention.**

# Zurück zur Grafik ...

Jetzt wissen wir ein wenig darüber, wie Events funktionieren (später erfahren wir noch mehr), und wollen uns wieder der Frage zuwenden, wie man Sachen auf dem Bildschirm platziert. Beschäftigen wir uns kurz mit ein paar netten Möglichkeiten, Grafiken zu erhalten, bevor wir zum Event-Handling zurückkehren.

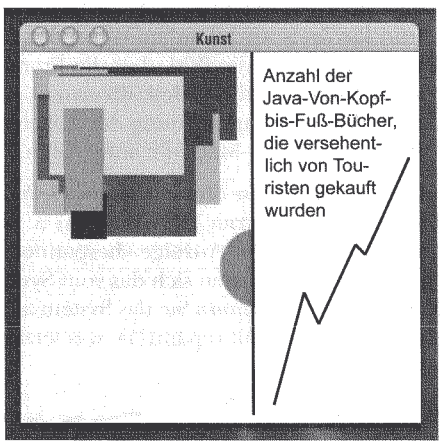
## Drei Arten, Dinge in eine GUI zu setzen:

### ① Widgets in einen Frame setzen

Fügen Sie Buttons, Menüs, Radio-Buttons usw. hinzu.

```
frame.getContentPane().add(meinButton);
```

Das Paket javax.swing hält mehr als ein Dutzend Widget-Typen bereit.




### ② 2-D-Grafiken auf einem Widget zeichnen

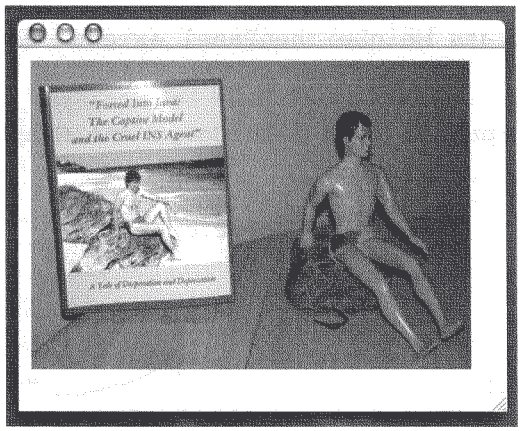
Zeichnen Sie mit einem Graphics-Objekt geometrische Figuren.

```
graphics.fillOval(70,70,100,100);
```

Sie können nicht nur Rechtecke und Kreise zeichnen, sondern noch vieles andere; die Java2D-API ist voll von tollen ausgeklügelten Grafikmethoden.

Kunst, Spiele, Simulationen usw.

Diagramme, Geschäftsgrafiken usw.



### ③ Ein JPEG auf ein Widget setzen

Sie können Ihre eigenen Bilder auf ein Widget setzen.

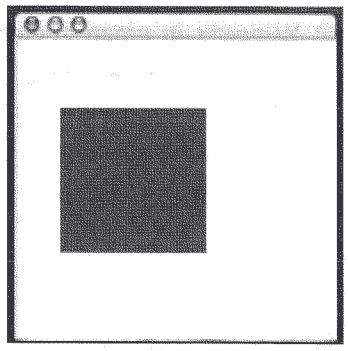
```
graphics.drawImage(meinBild,10,10,this);
```

# Machen Sie sich Ihr eigenes Grafik-Widget

Wenn Sie Ihre eigenen Grafiken auf den Bildschirm zaubern möchten, machen Sie sich am besten Ihr eigenes Widget, auf dem Sie zeichnen oder malen können. Dieses Widget setzen Sie in den Frame, genau wie einen Button oder irgendein anderes Widget, aber wenn es dort sichtbar wird, sind *Ihre* Bilder drauf. Sie können sogar bewegte Bilder erzeugen (in einer Animation) oder die Farben auf dem Bildschirm bei jedem Klick auf einen Button ändern lassen.

Es ist wirklich ein Kinderspiel.

## Machen Sie eine Unterklasse von JPanel und überschreiben Sie eine der Methoden, paintComponent().



Ihr gesamter Grafik-Code steht in der Methode paintComponent(). Sehen Sie paintComponent() als die Methode an, die vom System aufgerufen wird und gesagt bekommt: »He Widget, es ist Zeit, dass du dich zeichnest.« Wenn Sie einen Kreis zeichnen wollen, enthält die Methode paintComponent() Code für das Zeichnen eines Kreises. Wenn der Frame, der Ihr »Zeichenbrett« – das »Panel« – enthält, angezeigt wird, wird paintComponent() aufgerufen, und Ihr Kreis wird sichtbar. Wenn der Benutzer das Fenster ikonifiziert/minimiert, weiß die JVM, dass der Frame beim Wiederherstellen »repariert« werden muss, und ruft deshalb erneut paintComponent() auf. Wann immer die JVM eine Auffrischung der Anzeige für angebracht hält, wird Ihre paintComponent()-Methode aufgerufen.

Und noch eines: ***Sie rufen diese Methode niemals selbst auf!*** Das Argument dieser Methode (ein Objekt vom Typ Graphics) ist die eigentliche Leinwand zum Bemalen, mit der die *echte* Anzeige »bespannt« wird. Da kommen Sie selbst gar nicht dran – Sie müssen sich das vom System geben lassen. Wie Sie später noch sehen werden, *können* Sie das System allerdings darum bitten, die Anzeige aufzufrischen (mit repaint()), was letztendlich zum Aufruf von paintComponent() führt.

```
import java.awt.*;
import javax.swing.*;

class MeinZeichenPanel extends JPanel {

  public void paintComponent(Graphics g) {

    g.setColor(Color.orange);

    g.fillRect(20,50,100,100);
  }
}
```

Die brauchen Sie beide.

Schreiben Sie eine Unterklasse von JPanel, ein Widget, das Sie zu einem Frame hinzufügen können, genau wie alles andere. Mit dem Unterschied, dass dies Ihr eigenes maßgeschneidertes Widget ist.

Dies ist die MEGA-wichtige Graphics-Methode. Sie rufen sie NIEMALS selbst auf. Das System ruft sie auf und sagt: »Hier haben Sie eine schöne unbenutzte Maloberfläche vom Typ Graphics, auf der Sie zeichnen dürfen.«

Stellen Sie sich vor, »g« wäre eine Malmaschine. Sie sagen ihr, mit welcher Farbe sie malen soll und welche Form (mit Koordinaten für »wohin« und »wie groß«).

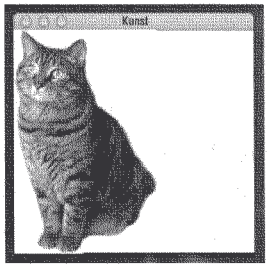
# Was man mit paintComponent() Schönes machen kann

Sehen wir uns noch ein paar weitere Dinge an, die Sie in paintComponent() machen können. Am meisten Spaß macht es aber, wenn Sie selbst experimentieren. Versuchen Sie, mit den Zahlen herumzuspielen, und sehen Sie sich die Klasse Graphics in der API an (wie wir später sehen werden, gibt es neben Graphics sogar noch *mehr* Möglichkeiten).

## Ein JPEG anzeigen

```
public void paintComponent(Graphics g) {

    Image image = new ImageIcon("catzilla.jpg").getImage();

    g.drawImage(image,3,4,this);

}
```

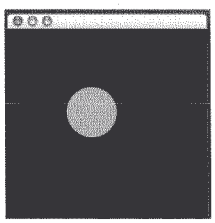
Hier steht Ihr Dateiname.




Hier stehen die (x,y)-Koordinaten für die obere linke Ecke des Bildes. Dies hier bedeutet »3 Pixel vom linken Rand des Panels und 4 Pixel vom oberen Rand des Panels entfernt«. Diese Zahlen beziehen sich immer auf das jeweilige Widget (in diesem Fall Ihre JPanel-Unterklasse), nicht auf den gesamten Frame.

---

## Einen Kreis in Zufallsfarben auf einen schwarzen Hintergrund malen



```
public void paintComponent(Graphics g) {

  g.fillRect(0,0,this.getWidth(), this.getHeight());

  int rot = (int) (Math.random() * 255);
  int grün = (int) (Math.random() * 255);
  int blau = (int) (Math.random() * 255);

  Color randomColor = new Color(rot, grün, blau);
  g.setColor(randomColor);
  g.fillOval(70,70,100,100);
}
```

Füllen Sie das gesamte Panel mit Schwarz (der voreingestellten Farbe).

Die ersten beiden Argumente definieren die (x,y)-Werte – bezogen auf das Panel – für die obere linke Ecke, in der mit dem Zeichnen begonnen wird. »0,0« heißt daher: »Beginne 0 Pixel vom linken Rand aus und 0 Pixel vom oberen Rand aus.« Die anderen beiden Argumente bedeuten: »Mach dieses Rechteck so breit wie das Panel (this.getWidth()) und so hoch wie das Panel (this.getHeight()).«

Sie können eine Farbe festlegen, indem Sie drei int-Zahlen für die RGB-Werte übergeben.

Beginne 70 Pixel von links, 70 von oben; mach es 100 Pixel breit und 100 hoch.

# Hinter jeder guten Graphics-Referenz steckt ein Graphics2D-Objekt

Das Argument für paintComponent() wird mit dem Typ Graphics (java.awt.Graphics) deklariert.

```
public void paintComponent(Graphics g) { }
```

Der Parameter »g« IST-EIN Graphics-Objekt. Was bedeutet, er *könnte* auch eine *Unter*klasse von Graphics sein (durch Polymorphie). Und in der Tat *ist* er das.

***Das vom Parameter »g« referenzierte Objekt ist eigentlich eine Instanz der Klasse* Graphics2D.**

Warum das überhaupt eine Rolle spielt? Weil es Dinge gibt, die Sie mit einer Graphics2D-Referenz machen können, mit einer Graphics-Referenz jedoch nicht. Ein Graphics2D-Objekt kann mehr als ein Graphics-Objekt, und hinter der Graphics-Referenz verbirgt sich in Wahrheit eine Graphics2D-Referenz.

Denken Sie polymorph. Welche Methoden Sie aufrufen können, entscheidet der Compiler anhand des Referenztyps, nicht anhand des Objekttyps. Wenn Sie ein Hund-Objekt haben, auf das von einer Tier-Referenzvariablen verwiesen wird:

```
Tier t = new Hund();
```

können Sie NICHT sagen:

```
t.bellen();
```

auch wenn Sie genau wissen, dass in Wirklichkeit ein Hund dahintersteckt. Der Compiler schaut sich »t« an, sieht, dass es vom Typ Tier ist, und findet auf der Fernbedienung für die Klasse Tier keine Taste für bellen(). Aber Sie können das Objekt immer noch in den Hund zurückverwandeln, der es tatsächlich ist, indem Sie schreiben:

```
Hund h = (Hund) t;
h.bellen();
```

Das Entscheidende beim Graphics-Objekt ist also:

**Wenn Sie eine Methode aus der Klasse Graphics2D benutzen wollen, können Sie den paintComponent-Parameter (»g«) *nicht* direkt in der Form benutzen, wie ihn die Methode übergeben bekommt. Sie können ihn aber mit einem Cast in eine neue Graphics2D-Variable *umwandeln*.**

```
Graphics2D g2d = (Graphics2D) g;
```

**Methoden, die Sie auf einer Graphics-Referenz aufrufen können:**

- drawImage()
- drawLine()
- drawPolygon()
- drawRect()
- drawOval()
- fillRect()
- fillRoundRect()
- setColor()

**Typumwandlung des Graphics2D-Objekts in eine Graphics2D-Referenz:**

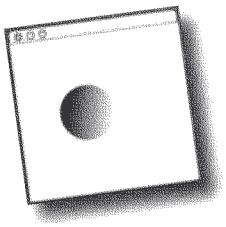
```
Graphics2D g2d = (Graphics2D) g;
```

**Methoden, die Sie auf einer Graphics2D-Referenz aufrufen können:**

- fill3DRect()
- draw3DRect()
- rotate()
- scale()
- shear()
- transform()
- setRenderingHints()

Diese Listen sind nicht vollständig; welche Methoden es noch gibt, können Sie in der API sehen.

# Weil das Leben zu kurz ist, um den Kreis mit einer einzigen Farbe auszumalen, wenn stattdessen ein wunderschöner Farbverlauf auf Sie wartet



```
public void paintComponent(Graphics g) {
  Graphics2D g2d = (Graphics2D) g;
  GradientPaint gradient = new GradientPaint(70,70,Color.blue, 150,150, Color.orange);
  g2d.setPaint(gradient);
  g2d.fillOval(70,70,100,100);
}
```

In Wirklichkeit ist das ein Graphics2D-Objekt, das als bloßes Graphics-Objekt verkleidet ist.

Machen wir einen Cast, damit wir etwas aufrufen können, das es nur in Graphics2D gibt, aber nicht in Graphics.

Startpunkt, Anfangsfarbe, Endpunkt, Endfarbe

Damit wird der virtuelle Farbpinsel auf einen Gradienten gesetzt statt auf eine einheitliche Farbe.

Die Methode fillOval() bedeutet eigentlich: »Fülle das Oval mit dem, womit dein Pinsel gerade geladen ist (d.h. mit dem Gradienten).«

---

```
public void paintComponent(Graphics g) {
  Graphics2D g2d = (Graphics2D) g;

  int rot = (int) (Math.random() * 255);
  int grün = (int) (Math.random() * 255);
  int blau = (int) (Math.random() * 255);
  Color startColor = new Color(rot, grün, blau);

  rot = (int) (Math.random() * 255);
  grün = (int) (Math.random() * 255);
  blau = (int) (Math.random() * 255);
  Color endColor = new Color(rot, grün, blau);

  GradientPaint gradient = new GradientPaint(70,70,startColor, 150,150, endColor);
  g2d.setPaint(gradient);
  g2d.fillOval(70,70,100,100);
}
```

Genau wie oben, abgesehen davon, dass für die Anfangs- und Endfarben des Gradienten Zufallsfarben erzeugt werden. Probieren Sie's aus!

# Punkt für Punkt

## EVENTS

- Um eine GUI zu erstellen, beginnen Sie mit einem Fenster, normalerweise einem JFrame:

```
JFrame frame = new JFrame();
```

- So können Sie Widgets (Buttons, Textfelder usw.) zum JFrame hinzufügen:

```
frame.getContentPane().add(button);
```

- Vor Java 5 konnten Sie – anders als bei den meisten anderen Komponenten – etwas nicht direkt zum JFrame hinzufügen, sondern mussten es zur ContentPane des JFrame hinzufügen. In Java 5 geht auch:

```
frame.add(button);
```

- Damit das Fenster (der JFrame) angezeigt wird, müssen Sie ihm eine Größe geben und ihm sagen, dass es sichtbar sein soll:

```
frame.setSize(300,300);
frame.setVisible(true);
```

- Um es zu erfahren, wenn der Benutzer den Button anklickt (oder irgendeine andere Aktion auf der Benutzeroberfläche durchführt), müssen Sie auf ein GUI-Event lauschen.
- Um auf ein Event zu lauschen, müssen Sie sich bei einer Event-Quelle registrieren. Eine Event-Quelle ist das Ding (Button, Kontrollkästchen usw.), das auf Grund der Benutzeraktion ein Event absetzt (oder »abfeuert«).
- Das Listener-Interface bietet der Event-Quelle eine Möglichkeit, Sie zu benachrichtigen. Zu diesem Zweck definiert es die Methode(n), die die Event-Quelle später bei einem Event aufruft.
- Um sich bei einer Quelle für Events zu registrieren, rufen Sie die Registrierungsmethode der Quelle auf. Registrierungsmethoden haben immer die folgende Form: ***add<Event-Typ>Listener.*** Um sich beispielsweise für die ActionEvents eines Buttons zu registrieren, rufen Sie auf:

```
button.addActionListener(this);
```

- Implementieren Sie das Listener-Interface, indem Sie all seine Event-Handling-Methoden implementieren. Schreiben Sie Ihren Code für das Event-Handling in die Methode, mit deren Hilfe der Listener Sie benachrichtigt. Bei ActionEvents ist es folgende Methode:

```
public void actionPerformed
                (ActionEvent event) {
  button.setText("Sie haben
                  geklickt!");
}
```

- Das Event-Objekt, das der Event-Handler-Methode übergeben wird, trägt Informationen über das Event, einschließlich der Quelle des Events.

## GRAPHICS

- 2-D-Grafiken können Sie direkt auf ein Widget zeichnen.
- Ein GIF oder ein JPEG können Sie direkt auf ein Widget zeichnen.
- Um Ihre eigenen Grafiken zu zeichnen (einschließlich GIF oder JPEG), machen Sie eine Unterklasse von JPanel und überschreiben die Methode paintComponent().
- Die Methode paintComponent() wird vom GUI-System aufgerufen. SIE RUFEN SIE NIEMALS SELBST AUF. Das Argument, das paintComponent() übergeben wird, ist ein Graphics-Objekt; dieses gibt Ihnen eine Oberfläche zum Zeichnen, die auf dem Bildschirm aufgespannt wird. Dieses Objekt können Sie nicht selbst erzeugen.
- Typische Methoden, die auf einem Graphics-Objekt (dem Parameter von paintComponent()) aufgerufen werden:

```
g.setColor(Color.blue);
g.fillRect(20,50,100,120);
```

- Um ein JPEG zu zeichnen, erstellen Sie ein Image-Objekt wie folgt:

```
Image image = new ImageIcon
("catzilla.jpg").getImage();
```

und zeichnen dann das Bild mit:

```
g.drawImage(image,3,4,this);
```

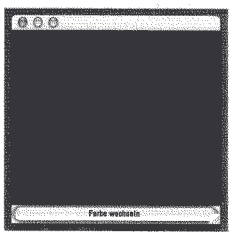
- Das Objekt, das von dem Graphics-Parameter der Methode paintComponent() referenziert wird, ist eigentlich eine Instanz der Klasse Graphics2D. Zur Klasse Graphics2D gehört eine Vielzahl von Methoden, unter anderem: fill3DRect(), draw3DRect(), rotate(), scale(), shear(), transform()
- Um die Methoden von Graphics2D aufzurufen, müssen Sie den Parameter mit einem Cast von einem Graphics-Objekt in ein Graphics2D-Objekt überführen:

```
Graphics2D g2d = (Graphics2D) g;
```

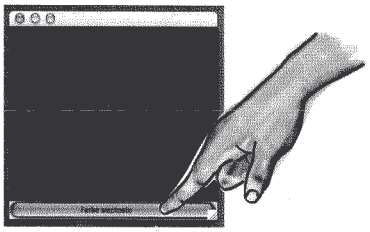
# Wir können ein Event erhalten.
# Und wir können Grafiken zeichnen.
# Aber können wir auch *genau dann* Grafiken zeichnen, wenn wir ein Event erhalten?

Lassen Sie uns ein Event mit einer Veränderung in unserem Zeichen-Panel verknüpfen. Wir lassen den Kreis bei jedem Klick auf den Button die Farbe wechseln. Hier der Programmablauf:

Anwendung starten

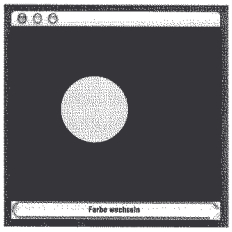



1. Der Frame mit den beiden Widgets (Ihrem Zeichen-Panel und einem Button) wird erstellt. Es wird ein Listener erzeugt und beim Button registriert. Dann wird der Frame angezeigt und wartet geduldig auf einen Klick des Benutzers.

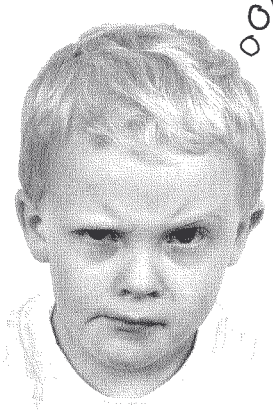


2. Der Benutzer klickt den Button an, der Button erzeugt ein Event-Objekt und ruft den Event-Handler des Listeners auf.

3. Der Event-Handler ruft repaint() auf dem Frame auf. Das System ruft paintComponent() auf dem Zeichen-Panel auf.




4. Voilà! Es wird eine neue Farbe gemalt, weil paintComponent() erneut ausgeführt wird. Der Kreis wird mit einer zufällig ausgewählten Farbe gefüllt.




# GUI-Layout: zwei oder mehr Widgets in einen Frame setzen

GUI-Layouts behandeln wir erst im *nächsten* Kapitel, aber wir geben Ihnen hier schon mal eine Kurzeinführung. Standardmäßig hat ein Frame fünf Bereiche, zu denen Sie etwas hinzufügen können. Zu jedem Bereich eines Frames können Sie nur *ein* Ding hinzufügen, aber das ist kein Grund zur Sorge! Denn dieses eine Ding könnte ein Panel sein, das drei weitere Dinge enthält, einschließlich eines Panels, das zwei weitere Dinge enthält ... Sie verstehen, wie es funktioniert? Eigentlich »mogeln« wir nämlich, wenn wir einen Button zum Frame auf folgende Weise hinzufügen:

```
frame.getContentPane().add(button);
```

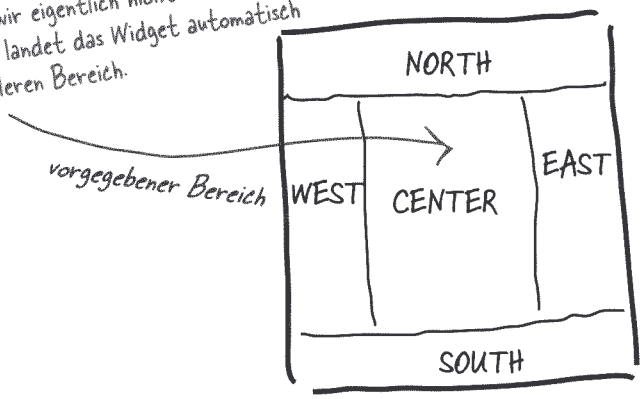
Dies ist der bessere (und normalerweise der vorschriftsmäßige) Weg, etwas zur Standard-ContentPane eines Frames hinzuzufügen. Geben Sie immer an, WO (in welchem Bereich) Sie das Widget haben wollen.

Auf diese Art (mit nur einem Argument für die add-Methode) sollte man es eigentlich nicht machen.

```
frame.getContentPane().add(BorderLayout.CENTER, button);
```

Wenn Sie die add-Methode aufrufen, die nur ein Argument annimmt (d.h. die, die wir eigentlich nicht benutzen sollten), landet das Widget automatisch im mittleren Bereich.

Wir rufen die add-Methode mit zwei Argumenten auf. Ihr wird ein Bereich übergeben (mittels einer Konstanten) sowie das Widget, das in diesen Bereich eingefügt werden soll.

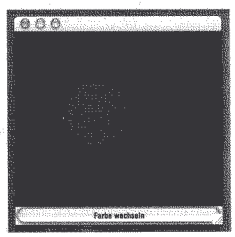



### Spitzen Sie Ihren Bleistift

Sehen Sie sich noch einmal die Abbildungen auf Seite 369 an und schreiben Sie dann den Code, der den Button und das Panel zum Frame hinzufügt.

## Der Kreis ändert bei jedem Klick auf den Button seine Farbe.

Das selbsterstellte Zeichen-Panel (eine Instanz von MeinZeichenPanel) sitzt im CENTER-Bereich des Frames.




Der Button liegt im SOUTH-Bereich des Frames.

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


public class EinfacheGui3C implements ActionListener {

  JFrame frame;

  public static void main (String[] args) {
    EinfacheGui3C gui = new EinfacheGui3C();
    gui.los();
  }

  public void los() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JButton button = new JButton("Farbe wechseln");
    button.addActionListener(this);

    MeinZeichenPanel zeichenPanel = new MeinZeichenPanel();

    frame.getContentPane().add(BorderLayout.SOUTH, button);
    frame.getContentPane().add(BorderLayout.CENTER, zeichenPanel);
    frame.setSize(300,300);
    frame.setVisible(true);
  }

  public void actionPerformed(ActionEvent event) {
    frame.repaint();
  }
}
```

Fügen Sie den Listener (this) zum Button hinzu.

Fügen Sie die beiden Widgets (Button und Zeichen-Panel) zu den beiden Bereichen des Frames hinzu.

Wenn der Benutzer klickt, sagen Sie dem Frame durch den Aufruf von repaint(), er solle sich neu zeichnen. Das bedeutet, dass paintComponent() auf jedem Widget in dem Frame aufgerufen wird!

---

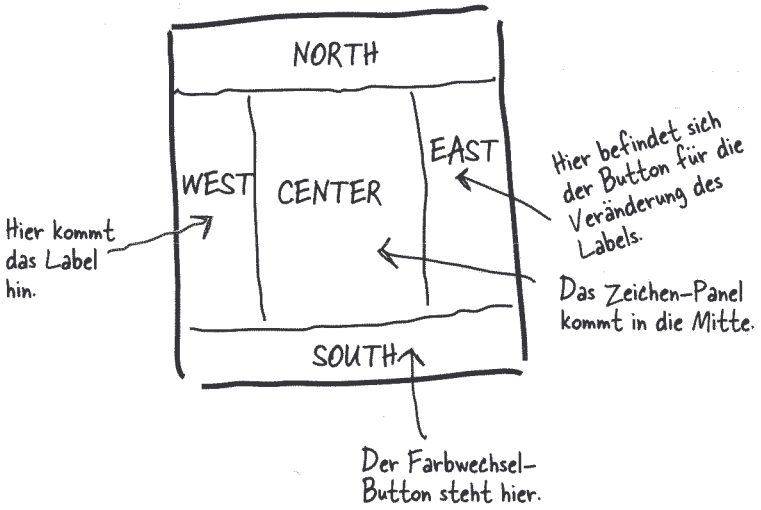
```
class MeinZeichenPanel extends JPanel {
  public void paintComponent(Graphics g) {
    // Code zum Ausfüllen des Ovals mit einer Zufallsfarbe
    // der Code steht auf Seite 367
  }

}
```

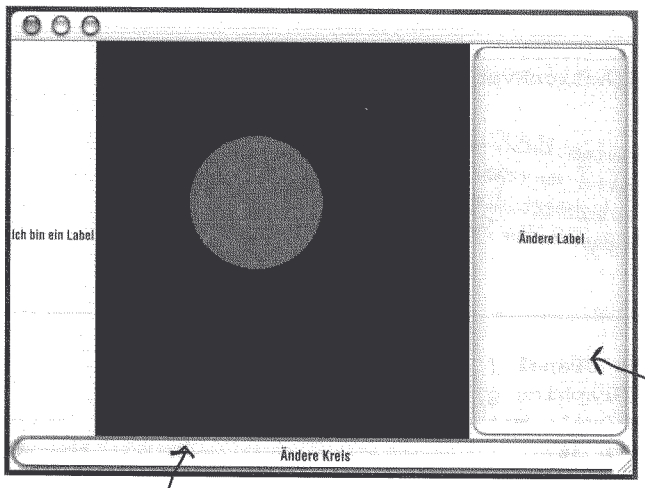
Die paintComponent()-Methode des Zeichen-Panels wird bei jedem Klick des Benutzers aufgerufen.

## Versuchen wir es mit ZWEI Buttons

Der Button im Süden wird sich so verhalten wie jetzt auch; er ruft einfach repaint() auf dem Frame auf. Der zweite Button (den wir in den östlichen Bereich setzen) verändert den Text auf einem Label. (Ein Label ist nichts anderes als ein Text auf dem Bildschirm.)

## Wir brauchen also jetzt VIER Widgets




## Und jetzt brauchen wir ZWEI Events

Oh ...

Ist das überhaupt möglich? Wie kommt man an *zwei* Events, wenn man nur *eine* actionPerformed()-Methode hat?

## Wie gelangen Sie an ActionEvents für zwei verschiedene Buttons, wenn jeder Button etwas anderes tun muss?

### ① *Möglichkeit 1:* Implementieren Sie zwei actionPerformed()-Methoden.

```
class MeineGui implements ActionListener {
  // hier steht erst eine Menge Code und dann:

  public void actionPerformed(ActionEvent event) {
    frame.repaint();
  }

  public void actionPerformed(ActionEvent event) {
    label.setText("Autsch!");
  }
}
```

Aber das ist unmöglich!

**Das Problem: Es geht nicht!** Sie können in einer Java-Klasse nicht zweimal die gleiche Methode implementieren. Das lässt sich nicht kompilieren. Und selbst wenn Sie es *könnten*, woher sollte die Event-Quelle wissen, *welche* der beiden Methoden sie aufrufen soll?

### ② *Möglichkeit 2:* Registrieren Sie den gleichen Listener bei beiden Buttons.

```
class MeineGui implements ActionListener {
  // hier wird ein Haufen von Instanzvariablen deklariert

  public void los() {
    // GUI erstellen
    colorButton = new JButton();
    labelButton = new JButton();
    colorButton.addActionListener(this);
    labelButton.addActionListener(this);
    // hier steht noch mehr GUI-Code ...
  }

  public void actionPerformed(ActionEvent event) {
    if (event.getSource() == colorButton) {
        frame.repaint();
    } else {
        label.setText("Autsch!");
    }
  }
}
```

Registrieren Sie den gleichen Listener bei beiden Buttons.

Befragen Sie das Event-Objekt, um herauszufinden, von welchem Button es abgesetzt wurde, und entscheiden Sie anhand dessen, was zu tun ist.

**Das Problem: Das funktioniert zwar, ist aber in den meisten Fällen nicht gerade gutes OO-Design!** Wenn ein Event-Handler viele verschiedene Dinge tut, bedeutet das, dass nur eine einzige Methode viele verschiedene Dinge tut. Und wenn Sie dann auch nur die Behandlung *einer* Event-Quelle verändern müssen, müssen Sie sich gleich mit der Event-Behandlung der halben Welt herumschlagen. Manchmal ist das tatsächlich eine gute Lösung, aber in der Regel kann man den Code dann nicht mehr so gut warten und erweitern.

## Wie gelangen Sie an ActionEvents für zwei verschiedene Buttons, wenn jeder Button etwas anderes tun muss?

### ③ *Möglichkeit 3:* Erstellen Sie zwei separate ActionListener-Klassen.

```
class MeineGui {
  JFrame frame;
  JLabel label;
  void gui() {
    // Code zur Instantiierung der beiden Listener und zur Registrierung
    // des einen beim Farb-Button und des anderen beim Label-Button
  }
} // Klasse schließen
```

---

```
class ColorButtonListener implements ActionListener {
  public void actionPerformed(ActionEvent event) {
    frame.repaint();
  }
}
```

Funktioniert nicht! Diese Klasse hat keine Referenz auf die Variable >>frame<< der Klasse MeineGui.

---

```
class LabelButtonListener implements ActionListener {
  public void actionPerformed(ActionEvent event) {
    label.setText("Autsch!");
  }
}
```

Ein Problem! Diese Klasse hat keine Referenz auf die Variable >>label<<.

**Das Problem: Diese Klassen haben keinen Zugriff auf die Variablen »frame« und »label«, mit denen sie etwas machen sollen!** Das könnten Sie zwar hinbiegen, aber dafür müssten Sie jeder der Listener-Klassen eine Referenz auf die Haupt-GUI-Klasse geben, damit der Listener innerhalb seiner actionPerformed()-Methode mit Hilfe dieser Referenz auf die Variablen der GUI-Klasse zugreifen könnte. Aber das bricht wiederum das Prinzip Kapselung, also müssten wir wahrscheinlich Getter-Methoden für die GUI-Widgets schreiben (getFrame(), getLabel() usw.). Und wahrscheinlich müssten Sie für die Listener-Klasse zusätzlich einen Konstruktor schreiben, damit Sie bei der Instantiierung des Listeners die GUI-Referenz an den Listener übergeben können. Und dann ... wird es immer verwirrender und komplizierter.

***Es muss einfach einen besseren Weg geben!***

Wäre es nicht wundervoll, wenn Sie zwei verschiedene Listener-Klassen hätten, aber diese Listener-Klassen könnten auf die Instanzvariablen der Haupt-GUI-Klasse zugreifen, fast so, als gehörten die Listener-Klassen *zu der anderen Klasse*? So könnten Sie beides unter einen Hut bringen.

Ja, das wäre fantastisch. Aber es ist ja nur ein Traum ...

# Innere Klassen kommen uns zu Hilfe

Sie können tatsächlich eine Klasse in eine andere einbetten. Es ist ganz einfach. Sie müssen nur die innere Klasse *innerhalb* der geschweiften Klammern der äußeren Klasse definieren.

### Einfache innere Klasse:

```
class MeineÄußereKlasse {

  class MeineInnereKlasse {
    void los() {
    }
  }

}
```

Die innere Klasse ist vollständig von der äußeren Klasse umschlossen.

Eine innere Klasse kann alle Methoden und Variablen der äußeren Klasse benutzen, sogar die privaten.

Die innere Klasse kann diese Variablen und Methoden genauso verwenden, als wären die Methoden und Variablen innerhalb der inneren Klasse deklariert.

Eine innere Klasse hat Sonderzutritt zu den Sachen ihrer äußeren Klasse. *Sogar zu den privaten Sachen.* Und die innere Klasse kann diese privaten Variablen und Methoden der äußeren Klasse so benutzen, als wären diese Member in der *inneren* Klasse definiert. Das ist das Praktische bei inneren Klassen – sie haben fast alle Vorteile einer normalen Klasse, aber zusätzlich spezielle Zugriffsrechte.

### Innere Klasse, die eine Variable der äußeren Klasse benutzt

```
class MeineÄußereKlasse {

  private int x;

  class MeineInnereKlasse {
    void los() {
      x = 42;
    }
  } // innere Klasse schließen

} // äußere Klasse schließen
```

Verwenden Sie »x«, als wäre es eine Variable der inneren Klasse.

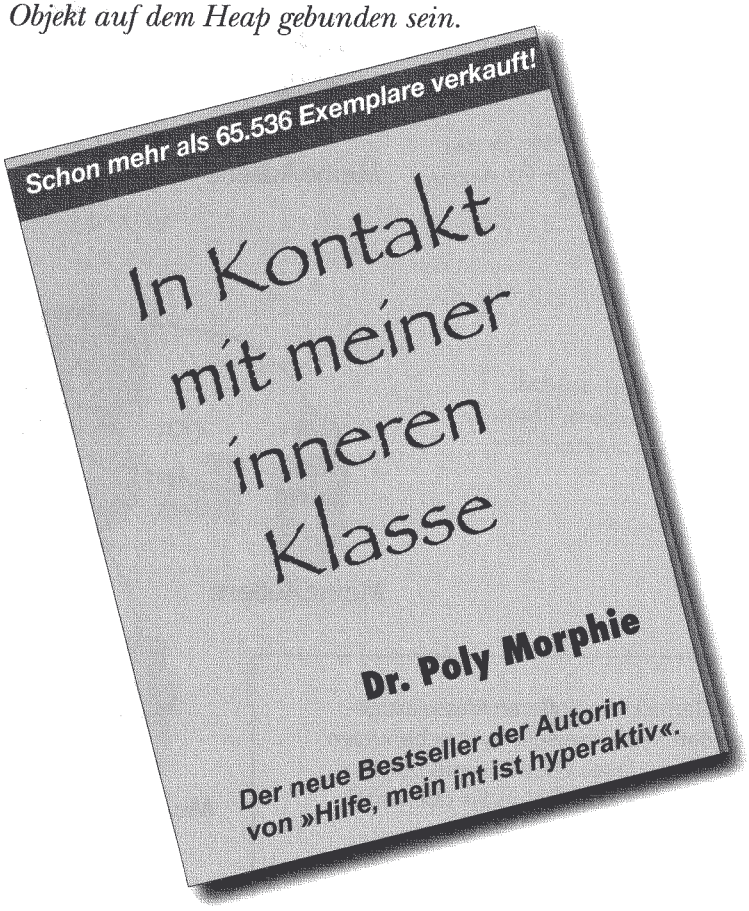
# Eine Instanz einer inneren Klasse muss an eine Instanz der äußeren Klasse gebunden sein.*

**Ein inneres Objekt hat eine ganz besondere Bindung an ein äußeres Objekt.**

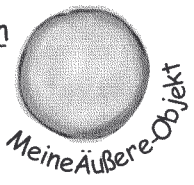
Denken Sie daran – wenn wir sagen, dass eine innere *Klasse* auf etwas in der äußeren Klasse zugreift, meinen wir eigentlich eine *Instanz* der inneren Klasse, die auf etwas in einer *Instanz* der äußeren Klasse zugreift. Aber in *welcher* Instanz?

Kann jede *beliebige* Instanz der inneren Klasse auf die Methoden und Variablen *jeder* Instanz der äußeren Klasse zugreifen?
**Nein!**

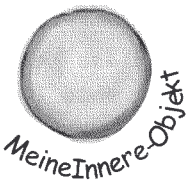
*Ein **inneres** Objekt muss an ein ganz bestimmtes **äußeres** Objekt auf dem Heap gebunden sein.*



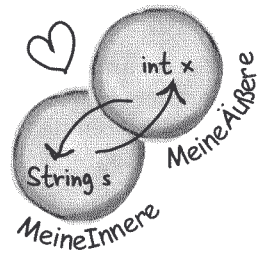

1. Erzeugen Sie eine Instanz der äußeren Klasse.




2. Erzeugen Sie mit Hilfe der Instanz der äußeren Klasse eine Instanz der inneren Klasse.




3. Das äußere und das innere Objekt sind jetzt innigst miteinander verbunden.




Diese beiden Objekte auf dem Heap haben eine ganz besondere Bindung aneinander. Das innere kann die Variablen des äußeren benutzen (und umgekehrt).

* Es gibt eine Ausnahme bei einem Sonderfall – einer inneren Klasse, die in einer statischen Methode definiert ist. Aber darauf gehen wir hier nicht ein, und es kann auch gut sein, dass Ihnen das in Ihrem gesamten Java-Leben nie begegnen wird.

# Wie man eine Instanz einer inneren Klasse erzeugt

Wenn Sie eine innere Klasse über Code instantiieren, der *innerhalb* einer äußeren Klasse steht, ist es die Instanz dieser äußeren Klasse, an die sich das innere Objekt »bindet«. Wenn beispielsweise Code innerhalb einer Methode die innere Klasse instantiiert, bindet sich das innere Objekt an die Instanz, deren Methode ausgeführt wird.

Der Code in einer äußeren Klasse kann eine der eigenen inneren Klassen auf genau die gleiche Weise instantiieren, wie er irgendeine andere Klasse instantiiert ... mit `new meineInnere()`:

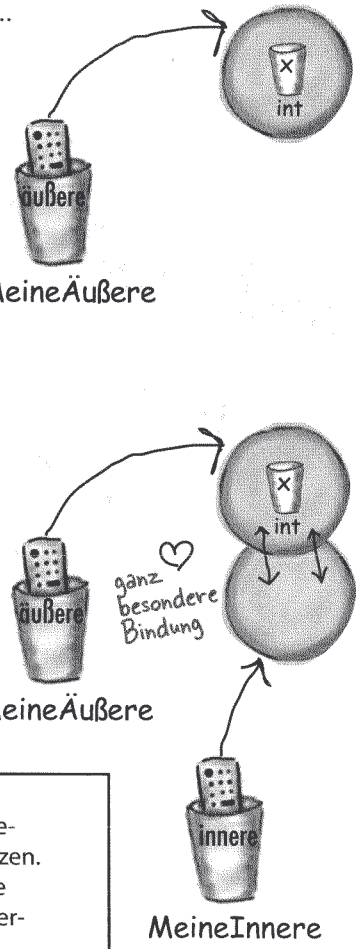
```
class MeineÄußere {

  private int x;

  MeineInnere meineInnere = new MeineInnere();

  public void tuWas() {
    meineInnere.los();
  }

  class MeineInnere {
    void los() {
      x = 42;
    }
  } // innere Klasse schließen

} // äußere Klasse schließen
```

Die äußere Klasse besitzt eine private Instanzvariable »x«.

Erzeugen Sie eine Instanz der inneren Klasse.

Rufen Sie eine Methode auf der inneren Klasse auf.

Die Methode in der inneren Klasse benutzt die Instanzvariable »x« der äußeren Klasse, als wäre »x« ein Element der inneren Klasse.

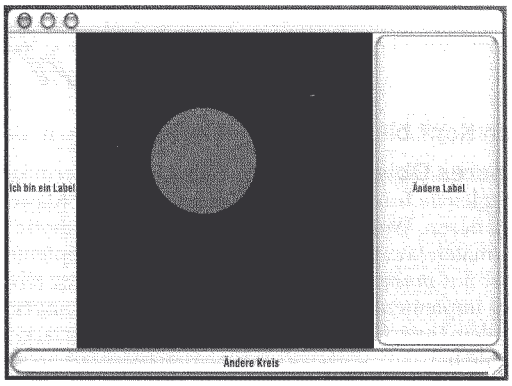



## Am Rande bemerkt

Sie *können* eine innere Instanz auch aus fremdem – außerhalb der äußeren Klasse ausgeführtem – Code heraus instantiieren, aber Sie müssen dafür eine spezielle Syntax benutzen. Die Chancen stehen allerdings gut, dass Sie durch Ihr ganzes Java-Leben kommen, ohne eine einzige innere Klasse von außerhalb erzeugen zu müssen, aber falls es Sie doch interessieren sollte ...

```
class RaritätenKabinett {
  public static void main (String[] args) {
    MeineÄußere äußeresObj = new MeineÄußere();
    MeineÄußere.MeineInnere inneresObj = äußeresObj.new MeineInnere();
  }
}
```

## Jetzt können wir den Code mit den zwei Buttons zum Laufen bringen

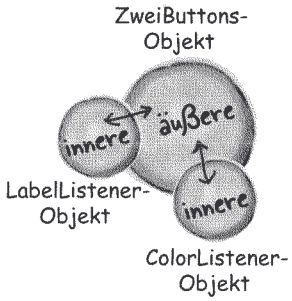



```
public class ZweiButtons {
```

Die Haupt-GUI-Klasse implementiert ActionListener jetzt nicht mehr.

```
  JFrame frame;
  JLabel label;

  public static void main (String[] args) {
    ZweiButtons gui = new ZweiButtons ();
    gui.los();
  }

  public void los() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JButton labelButton = new JButton("Ändere Label");
    labelButton.addActionListener(new LabelListener());

    JButton colorButton = new JButton("Ändere Kreis");
    colorButton.addActionListener(new ColorListener());

    label = new JLabel("Ich bin ein Label");
    MeinZeichenPanel zeichenPanel = new MeinZeichenPanel();

    frame.getContentPane().add(BorderLayout.SOUTH, colorButton);
    frame.getContentPane().add(BorderLayout.CENTER, zeichenPanel);
    frame.getContentPane().add(BorderLayout.EAST, labelButton);
    frame.getContentPane().add(BorderLayout.WEST, label);

    frame.setSize(420,300);
    frame.setVisible(true);
  }
```

Anstatt der Listener-Registrierungsmethode des Buttons this zu übergeben, übergeben Sie jetzt eine neue Instanz der passenden Listener-Klasse.




Jetzt haben wir endlich ZWEI ActionListener in einer einzigen Klasse!

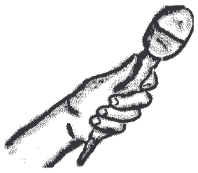
```
  class LabelListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
      label.setText("Autsch!");
    }
  } // innere Klasse schließen
```

Die innere Klasse kennt das >>label<<.

```
  class ColorListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
      frame.repaint();
    }
  } // innere Klasse schließen
```

Die innere Klasse kann jetzt die Instanzvariable >>frame<< benutzen, ohne eine explizite Referenz auf das Objekt der äußeren Klasse zu haben.

```
}
```



## Java im Gespräch

### Interview der Woche: Instanz einer inneren Klasse

**Von Kopf bis Fuß:** Warum sind innere Klassen wichtig?

**Inneres Objekt:** Wo soll ich anfangen? Wir ermöglichen es, das gleiche Interface mehr als einmal in einer Klasse zu implementieren. Wie Sie wissen, kann man eine Methode in einer normalen Java-Klasse nicht mehr als einmal implementieren. Aber mit *inneren* Klassen kann jede innere Klasse *dasselbe* Interface implementieren, und so werden all diese *unterschiedlichen* Implementierungen für ein- und dieselbe Interface-Methode möglich.

**Von Kopf bis Fuß:** Warum sollte man denn überhaupt jemals die gleiche Methode zweimal implementieren wollen?

**Inneres Objekt:** Betrachten wir noch einmal GUI-Event-Handler. Überlegen Sie ... wenn Sie *drei* Buttons haben, von denen jeder ein anderes Verhalten zeigen soll, dann benutzen Sie *drei* innere Klassen, die alle den ActionListener implementieren – was bedeutet, dass jede innere Klasse ihre eigene actionPerformed()-Methode implementiert.

**Von Kopf bis Fuß:** Also sind Event-Handler der einzige Grund für die Verwendung innerer Klassen?

**Inneres Objekt:** Oh nein, ganz und gar nicht! Event-Handler sind nur ein nahe liegendes Beispiel. Jedes Mal, wenn Sie eine separate Klasse brauchen, die sich aber trotzdem so verhalten soll, als wäre sie Teil einer *anderen* Klasse, ist eine innere Klasse die *beste* – und manchmal die *einzige* – Lösung.

**Von Kopf bis Fuß:** Ich verstehe das immer noch nicht ganz. Wenn Sie möchten, dass sich die innere Klasse so *verhält*, als gehöre sie zur äußeren Klasse, warum braucht man dann überhaupt erst eine separate Klasse? Warum könnte denn nicht der Code der inneren Klasse gleich *in* der äußeren Klasse stehen?

**Inneres Objekt:** Ich habe Ihnen doch gerade eine Situation *geschildert*, in der Sie mehr als eine Implementierung eines Interface brauchen. Doch selbst wenn Sie keine Interfaces benutzen, brauchen Sie möglicherweise zwei verschiedene *Klassen*, weil diese Klassen zwei verschiedene *Dinge* repräsentieren. Das ist gute OO-Programmierung.

**Von Kopf bis Fuß:** Halt, warten Sie doch mal. Ich dachte, bei einem Großteil des OO-Designs geht es um Wiederverwendung und Wartbarkeit. Wissen Sie, diese Idee, dass Sie bei zwei separaten Klassen beide Klassen unabhängig voneinander modifizieren und verwenden können statt alles in eine einzige Klasse zu stopfen, bla bla bla ... Aber bei einer *inneren* Klasse arbeiten Sie doch letztendlich noch immer mit nur einer einzigen *echten* Klasse, oder? Die umschließende Klasse ist die einzige, die wiederverwendbar und von allen anderen getrennt ist. Innere Klassen sind eigentlich nicht wiederverwendbar. Tatsächlich habe ich schon gehört, wie man sie als »wieder verwendungslos« bezeichnet hat – also als »immer wieder verwendungslos«.

**Inneres Objekt:** Ja, es stimmt, dass die innere Klasse nicht so gut wiederverwendbar ist, manchmal tatsächlich gar nicht, weil sie eng an die Instanzvariablen und Methoden der äußeren Klasse gebunden ist. Aber sie –

**Von Kopf bis Fuß:** – genau das meinte ich! Wenn sie nicht wiederverwendbar ist, warum soll man sich dann mit einer zusätzlichen Klasse abgeben? Ich meine, abgesehen von dem Interface-Problem, das sich für mich nach einer Behelfslösung anhört.

**Inneres Objekt:** Wie ich schon sagte, Sie müssen an IST-EIN-Beziehungen und Polymorphie denken.

**Von Kopf bis Fuß:** Okay. Und ich denke daran, weil ...

**Inneres Objekt:** Weil die äußeren und die inneren Klassen möglicherweise *unterschiedliche* IST-EIN-Tests bestehen müssen! Beginnen wir mit dem Beispiel des polymorphen GUI-Listener. Welcher Parametertyp wird für die Listener-Registrierungsmethode des Buttons deklariert? Anders ausgedrückt, wenn Sie in der API nachsehen, was für ein *Ding* müssen Sie der Methode addActionListener() übergeben? Von welchem Typ (Klasse oder Interface) muss es sein?

**Von Kopf bis Fuß:** Sie müssen einen Listener übergeben. Etwas, das ein bestimmtes Listener-Interface implementiert, in diesem Fall ActionListener. Aber das wissen wir doch alles schon. Worauf wollen Sie hinaus?

**Inneres Objekt:** Ich will darauf hinaus, dass Sie eine Methode haben, die – im Rahmen der Polymorphie – nur einen bestimmten *Typ* entgegennimmt. Etwas, das den IST-EIN-Test für ActionListener besteht. Aber – und jetzt kommt's – was ist, wenn Ihre Klasse ein IST-EIN von irgendetwas sein muss, das vom Typ her eine *Klasse* ist und kein Interface?

**Von Kopf bis Fuß:** Würden Sie Ihre Klasse nicht einfach die Klasse *erweitern* lassen, von der sie ein Teil sein muss? Ist das nicht genau die Art und Weise, wie Unterklassen funktionieren? Wenn B eine Unterklasse von A ist, kann überall dort, wo ein A erwartet wird, ein B verwendet werden. Diese ganze Übergib-einen-Hund-wo-ein-Tier-erwartet-wird-Geschichte.

**Inneres Objekt:** Ja! Bingo! Und was ist nun, wenn Sie den IST-EIN-Test für zwei verschiedene Klassen bestehen müssen? Klassen, die nicht zur gleichen Vererbungshierarchie gehören?

**Von Kopf bis Fuß:** Oh, stimmt, Sie können ... hmmm. Ich begreife allmählich, glaube ich. Sie können zwar mehr als nur ein Interface *implementieren*, aber Sie können immer nur *eine* Klasse *erweitern*. Wenn der Typ eine Klasse sein soll, kann man nur *eine* Art von IST-EIN sein.

**Inneres Objekt:** Sehr gut! Ja, Sie können nicht sowohl Hund als auch Button sein. Aber wenn Sie ein Hund sind, der manchmal ein Button sein muss (um sich selbst an Methoden zu übergeben, die einen Button erwarten), kann die Klasse Hund (die Tier erweitert und deshalb nicht Button erweitern kann) eine *innere* Klasse besitzen, die Button erweitert und im Namen des Hundes als Button auftritt. Und so kann der Hund immer dann, wenn ein Button verlangt wird, seinen inneren Button übergeben anstatt sich selbst. Anders ausgedrückt: Anstatt zu sagen x.nimmButton(this), ruft das Hund-Objekt x.nimmButton(new MeinInnererButton()) auf.

**Von Kopf bis Fuß:** Haben Sie vielleicht ein anschauliches Beispiel?

**Inneres Objekt:** Erinnern Sie sich an das Zeichen-Panel, das wir benutzt haben, als wir unsere eigene Unterklasse von JPanel erzeugt haben? Im Moment ist diese Klasse eine separate, nicht innere Klasse. Und das ist auch in Ordnung, weil die Klasse keinen speziellen Zugriff auf die Instanzvariablen der Haupt-GUI braucht. Aber was wäre, wenn sie ihn bräuchte? Was ist, wenn wir auf diesem Panel eine Animation erstellen und es die Koordinaten dafür von der Hauptanwendung bekommt (z.B. basierend auf irgendetwas, das der Benutzer an einer anderen Stelle in der GUI macht)? Wenn wir nun aus dem Zeichen-Panel eine innere Klasse machen, dann machen wir das Zeichen-Panel zu einer Unterklasse von JPanel, während die äußere Klasse weiterhin eine Unterklasse von irgendetwas anderem sein darf.

**Von Kopf bis Fuß:** Ja, ich verstehe! Und das Zeichen-Panel wäre als separate Klasse sowieso nicht wiederverwendbar genug, denn das, was es dann schließlich zeichnet, ist typisch für diese spezielle GUI-Anwendung.

**Inneres Objekt:** Ja, Sie haben es begriffen!

**Von Kopf bis Fuß:** Gut. Dann können wir jetzt zur Art der *Beziehung* zwischen Ihnen und der äußeren Instanz kommen.

**Inneres Objekt:** Was ist bloß los mit euch Leuten? Gibt es nicht genug schmutzige Klatschgeschichten bei einem ernsten Thema wie Polymorphie?

**Von Kopf bis Fuß:** Ach, Sie glauben ja gar nicht, wie viel die Öffentlichkeit für guten alten Boulevardblatt-Schmutz bezahlt. Also: Irgendjemand erzeugt Sie, und Sie werden sofort an das äußere Objekt gebunden, richtig?

**Inneres Objekt:** Ja, richtig. Und, ja: Man hat das auch schon mit einer arrangierten Heirat verglichen. Wir haben kein Mitspracherecht, wenn es darum geht, an welches Objekt wir gebunden werden.

**Von Kopf bis Fuß:** Gut, ich greife mal die Heiratsanalogie auf. Können Sie *sich scheiden* lassen und dann *jemand anderen* heiraten?

**Inneres Objekt:** Nein, es ist fürs Leben.

**Von Kopf bis Fuß:** Für wessen Leben? Für Ihres? Das des äußeren Objekts? Ihrer beider Leben?

**Inneres Objekt:** Meins. Ich kann nicht an ein anderes äußeres Objekt gebunden werden. Mein einziger Weg hinaus führt über die Garbage Collection.

**Von Kopf bis Fuß:** Was ist mit dem äußeren Objekt? Kann es sich mit anderen inneren Objekten zusammentun?

**Inneres Objekt:** Da wären wir also beim Thema. *Genau darauf* wollten Sie doch hinaus. Ja, ja. Mein so genannter »Lebensgefährte« kann so viele innere Objekte haben, wie er will.

**Von Kopf bis Fuß:** Ist das so was wie serielle Monogamie? Oder kann er sie alle zur gleichen Zeit haben?

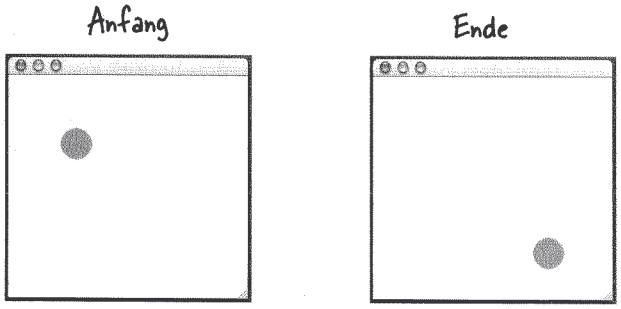
**Inneres Objekt:** Alle zur gleichen Zeit. Sind Sie nun zufrieden?

**Von Kopf bis Fuß:** Ja, das passt ins Bild. Und vergessen wir nicht – *Sie selbst* hatten ja die Vorzüge der »Mehrfachimplementierung des gleichen Interface« gerühmt. Es ist also einleuchtend, dass eine äußere Klasse mit drei Buttons auch drei verschiedene innere Klassen braucht (und somit drei verschiedene Objekte von inneren Klassen), um die Events zu behandeln. Ich bedanke mich vielmals. Hier haben Sie ein Taschentuch.

# Animation mit Hilfe einer inneren Klasse

Wie wir gesehen haben, sind innere Klassen bei Event-Listenern praktisch, weil Sie die gleiche Event-Handling-Methode mehr als einmal implementieren müssen. Aber jetzt werden wir uns einmal ansehen, wie nützlich eine innere Klasse als Unterklasse von etwas sein kann, was die äußere Klasse *nicht* erweitert. Anders ausgedrückt: wenn die äußere Klasse und die innere Klasse zu verschiedenen Vererbungsbäumen gehören!

Unser Ziel ist eine einfache Animation, bei der sich der Kreis von oben links nach unten rechts über den Bildschirm bewegt.




## Wie eine einfache Animation funktioniert

1. Zeichnen Sie ein Objekt mit bestimmten x- und y-Koordinaten.

```
g.fillOval(20,50,100,100);
```

20 Pixel von links, 50 Pixel von oben

2. Zeichnen Sie das Objekt erneut bei anderen x- und y-Koordinaten.

```
g.fillOval(25,55,100,100);
```

25 Pixel von links, 55 Pixel von oben

(das Objekt hat sich ein wenig nach unten rechts bewegt)

3. Wiederholen Sie den vorhergehenden Schritt mit den sich ändernden x- und y-Werten so lange, wie die Animation dauern soll.

### Es gibt keine Dummen Fragen

**F: Warum lernen wir hier etwas über Animation? Ich bezweifle, dass ich jemals Spiele schreiben werde.**

A: Auch wenn Sie keine Spiele schreiben werden, werden Sie doch vielleicht Simulationen erzeugen, in denen sich Dinge mit der Zeit verändern und so das Ergebnis eines Prozesses anzeigen. Oder vielleicht werden Sie ein Visualisierungswerkzeug entwickeln, das beispielsweise eine Grafik aktualisiert und anzeigt, wie viel Speicherplatz ein Programm verbraucht oder wie viel Datenverkehr durch Ihren Lastausgleichsserver fließt. Irgendetwas, das einen Satz von sich kontinuierlich ändernden Zahlen entgegennehmen muss und ihn in irgendetwas Nützliches übersetzt, um den Zahlen Informationen zu entnehmen.

Klingt das nicht alles äußerst seriös? Natürlich ist das nur die »offizielle Rechtfertigung«. *Eigentlich* behandeln wir das hier, weil man damit eine weitere Anwendungsmöglichkeit für innere Klassen auf einfache Weise demonstrieren kann. (Und weil wir Animation einfach *lieben* und unser nächstes Buch von J2EE handelt und wir genau *wissen*, dass wir da keine Animationen reinbringen können.)

# Eigentlich wollen wir etwas in dieser Art ...

```
class MeinZeichenPanel extends JPanel {
  public void paintComponent(Graphics g) {
    g.setColor(Color.orange);
    g.fillOval(x,y,100,100);
  }
}
```

Bei jedem Aufruf von paintComponent() wird das Oval an eine andere Stelle gezeichnet.

Spitzen Sie Ihren Bleistift

## Aber wie kommen wir an die neuen x- und y-Koordinaten?

## Und wer ruft repaint() auf?

Versuchen Sie, ob Sie eine **einfache Lösung entwerfen** können, bei der sich der Ball von der oberen linken Ecke des Zeichen-Panels nach unten rechts bewegt. Unsere Antwort steht auf der nächsten Seite – blättern Sie also nicht um, bevor Sie fertig sind!

Tipp (eher gesagt, Wink mit dem Zaunpfahl): Machen Sie aus dem Zeichen-Panel eine innere Klasse.

Noch ein Tipp: Verwenden Sie in der Methode paintComponent() keinerlei Wiederholungsschleifen.

**Schreiben Sie Ihre Ideen (oder den Code) hier auf:**

## Der vollständige Code für die einfache Animation

```
import javax.swing.*;
import java.awt.*;

public class EinfacheAnimation {

  int x = 70;
  int y = 70;
```

Erstellen Sie zwei Instanzvariablen in der Haupt-GUI-Klasse für die x- und die y-Koordinate des Kreises.

```
  public static void main (String[] args){
    EinfacheAnimation gui = new EinfacheAnimation();
    gui.los();
  }

  public void los(){
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    MeinZeichenPanel zeichenPanel = new MeinZeichenPanel();

    frame.getContentPane().add(zeichenPanel);
    frame.setSize(300,300);
    frame.setVisible(true);
```

Erzeugen Sie die Widgets wie gehabt und setzen Sie sie in den Frame.

Und hier geht's dann richtig los!

```
    for (int i = 0; i < 130; i++){

      x++;
      y++;

      zeichenPanel.repaint();

      try {
        Thread.sleep(50);
      } catch (Exception e){ }
    }

  } // Methode los() schließen
```

Wiederholen Sie das 130-mal.

Erhöhen Sie die x- und y-Koordinaten

Sagen Sie dem Panel, es soll sich erneut zeichnen (damit wir den Kreis an seinem neuen Platz sehen können).

Verlangsamen Sie das Ganze ein bisschen (sonst bewegt es sich so schnell, dass Sie die Bewegung gar nicht SEHEN). Keine Sorge – das ist nichts, was Sie schon hätten wissen müssen. Zu Threads kommen wir erst in Kapitel 15.

Jetzt ist das eine innere Klasse.

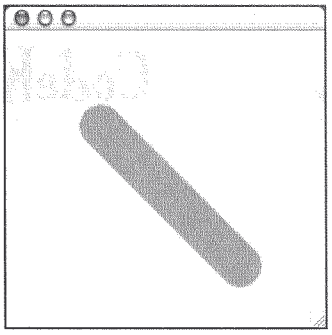
```
  class MeinZeichenPanel extends JPanel {

    public void paintComponent(Graphics g) {
      g.setColor(Color.green);
      g.fillOval(x,y,40,40);

    }
  } // innere Klasse schließen
} // äußere Klasse schließen
```

Benutzen Sie hier die kontinuierlich aktualisierten x- und y-Koordinaten der äußeren Klasse.

## Oh-oh! Er hat sich nicht bewegt ... er *verschmiert!*

Was haben wir falsch gemacht?

In der paintComponent()-Methode steckt ein kleines Problem.



Nicht ganz das, was wir haben wollten.

### Wir haben vergessen auszuradieren, was schon da stand! Deshalb wurde eine Spur gezogen.

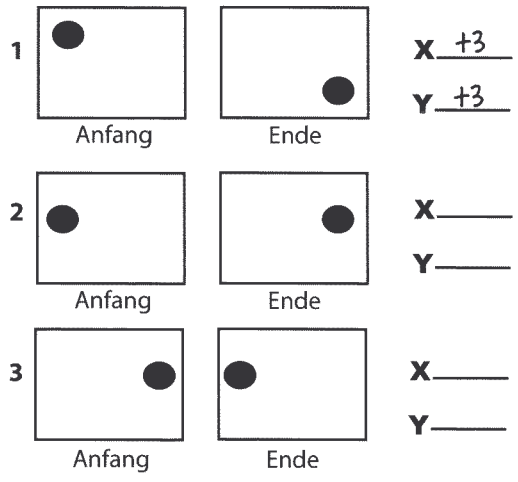
Um das in Ordnung zu bringen, müssen wir einfach nur jedes Mal das gesamte Panel mit der Hintergrundfarbe füllen, bevor wir den Kreis zeichnen. Im unten stehenden Code sind am Anfang der Methode zwei Zeilen hinzugefügt worden: eine, die die Farbe auf Weiß setzt (die Hintergrundfarbe des Zeichen-Panels), und die andere, die das gesamte Panel-Rechteck mit dieser Farbe füllt. Auf Deutsch besagt der unten stehende Code: »Fülle ein Rechteck beginnend bei x gleich 0 und y gleich 0 (0 Pixel von links und 0 Pixel von oben), und zwar so breit und so hoch, wie das Panel derzeit ist.«

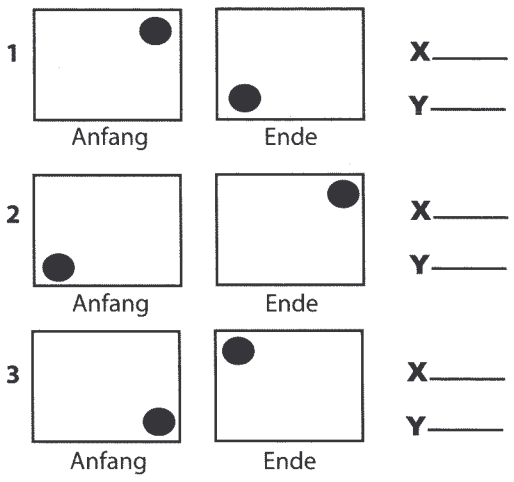
```
public void paintComponent(Graphics g) {
  g.setColor(Color.white);
  g.fillRect(0,0,this.getWidth(), this.getHeight());

  g.setColor(Color.green);
  g.fillOval(x,y,40,40);

}
```

getWidth() und getHeight() sind von JPanel geerbte Methoden.

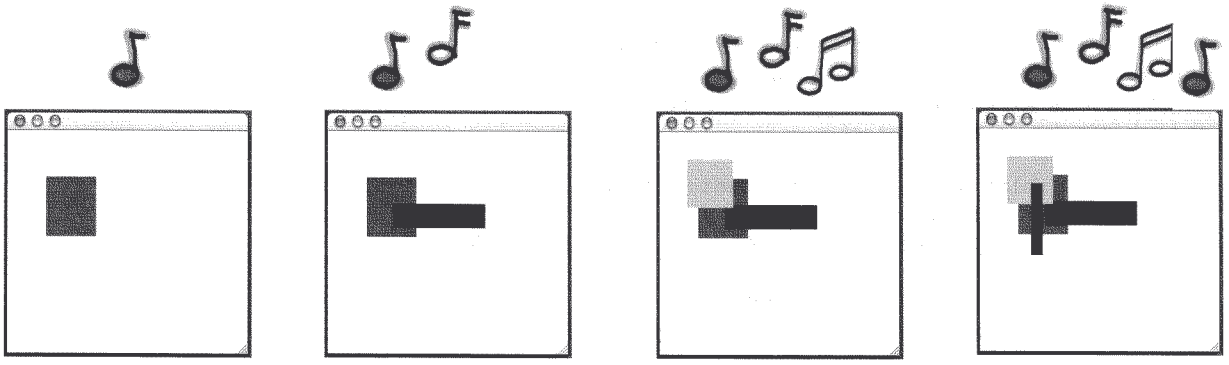
## Spitzen Sie Ihren Bleistift (freiwillig, nur zum Spaß)

Auf welche Weise würden Sie die x- und y-Koordinaten verändern, um die unten abgebildeten Animationen zu erhalten? (Nehmen Sie an, in dem ersten Beispiel findet eine Bewegung in 3-Pixel-Schritten statt.)



# CodeKüche



Schlag eins

Schlag zwei

Schlag drei

Schlag vier ...

**Machen wir ein Musikvideo! Wir benutzen dazu von Java zufallsgenerierte Grafiken, die im Takt der Musik erscheinen.**

**Ganz nebenbei werden wir uns auch für eine neue Art von Nicht-GUI-Event – ausgelöst von der Musik selbst – registrieren (und darauf lauschen).**

Sie wissen ja, dieser ganze Teil ist optional. Aber wir glauben, dass er gut für Sie ist. Und er wird Ihnen gefallen. Und Sie können andere Menschen damit beeindrucken.

(Okay, sicher, das funktioniert vielleicht nur, wenn sie leicht zu beeindrucken sind, aber trotzdem ...)

# Auf ein Nicht-GUI-Event lauschen

Gut, es wird vielleicht nicht gerade ein Musikvideo, aber auf jeden Fall ein Programm, das im Takt der Musik zufällig erzeugte Grafikelemente auf den Bildschirm malt. Kurz gesagt, das Programm lauscht auf den Takt der Musik und zeichnet bei jedem Schlag ein zufallsabhängiges Rechteck.

Das beschert uns ein paar neue Probleme. Bis jetzt haben wir nur auf GUI-Events gelauscht, aber jetzt müssen wir auf eine bestimmte Art von MIDI-Event lauschen. Das Lauschen auf ein Nicht-GUI-Event funktioniert jedoch genauso wie das Lauschen auf ein GUI-Event: Sie implementieren ein Listener-Interface, registrieren den Listener bei einer Event-Quelle, lehnen sich dann zurück und warten darauf, dass die Event-Quelle Ihre Event-Handler-Methode aufruft (die im Listener-Interface definierte Methode).

Die einfachste Möglichkeit, auf den Takt der Musik zu lauschen, wäre es, sich bei den eigentlichen MIDI-Events zu registrieren und auf diese zu lauschen. Jedes Mal, wenn der Sequencer das Event erhält, würde dann auch Ihr Code das Event erhalten und könnte die Grafik zeichnen. Aber ... da gibt es ein Problem. Genauer gesagt, einen Bug, der verhindert, dass wir auf die MIDI-Events lauschen können, die *wir selbst* erzeugen (die für NOTE ON).

Wir müssen also ein bisschen tricksen. Es gibt noch eine andere Art von MIDI-Event, auf das wir lauschen können und das ControllerEvent heißt. Unsere Lösung besteht darin, uns für ControllerEvents zu registrieren und dann dafür zu sorgen, dass bei jedem NOTE ON-Event zum selben »Schlag« ein zugehöriges ControllerEvent abgesetzt wird. Wie wir sicherstellen können, dass das ControllerEvent zur gleichen Zeit abgesetzt wird? Wir geben es – genau wie die anderen Events – zum Track hinzu! Mit anderen Worten, unsere Musiksequenz sieht so aus:

SCHLAG 1 – NOTE ON, CONTROLLER-EVENT

SCHLAG 2 – NOTE OFF

SCHLAG 3 – NOTE ON, CONTROLLER-EVENT

SCHLAG 4 – NOTE OFF

usw.

Bevor wir in das volle Programm einsteigen, wollen wir aber das Erzeugen und Hinzufügen von MIDI-Messages/-Events noch ein bisschen vereinfachen, denn in *diesem* Programm werden wir wirklich viele davon erzeugen.

## Was unser Programm tun muss:

1. Eine Reihe von MIDI-Messages/-Events machen, die zufällige Töne auf einem Klavier spielen (oder einem beliebigen Instrument Ihrer Wahl).
2. Einen Listener für die Events registrieren.
3. Den Sequencer starten.
4. Bei jedem Aufruf der Event-Handler-Methode des Listener ein zufälliges Rechteck auf das Zeichen-Panel zeichnen und repaint() aufrufen.

## Wir gehen in drei Iterationsschritten vor

1. Version 1: Code, der das Erzeugen und Hinzufügen von MIDI-Events vereinfacht, weil wir eine Menge von diesen brauchen werden.
2. Version 2: Registrierung und Lauschen auf die Events, aber noch ohne Grafik. Gibt bei jedem Schlag eine Meldung auf der Kommandozeile aus.
4. Version 3: Die marktreife Fassung. Ergänzt Version 2 um grafische Elemente.

# Ein einfacherer Weg zur Erzeugung von Messages/Events

Im Moment ist das Erzeugen und Hinzufügen von Messages zu einem Track mühsam. Für jede Message müssen wir die Message-Instanz erzeugen (in diesem Fall ShortMessage), setMessage() aufrufen, ein MidiEvent für die Message machen und das Event zum Track hinzufügen. Im Code des letzten Kapitels sind wir bei jeder Message sämtliche Schritte durchgegangen. Das bedeutet acht Zeilen Code, nur damit ein Ton erklingt und dann wieder aufhört! Vier Zeilen, um ein NOTE ON-Event hinzuzufügen, und vier Zeilen, um ein NOTE OFF-Event hinzuzufügen.

```
ShortMessage a = new ShortMessage();
a.setMessage(144, 1, note, 100);
MidiEvent noteOn = new MidiEvent(a, 1);
track.add(noteOn);

ShortMessage b = new ShortMessage();
b.setMessage(128, 1, note, 100);
MidiEvent noteOff = new MidiEvent(b, 16);
track.add(noteOff);
```

### Was bei jedem Event passieren muss:

1. Eine Message-Instanz erzeugen.

   ```
   ShortMessage a = new ShortMessage();
   ```

2. setMessage() mit den Anweisungen aufrufen.

   ```
   a.setMessage(144, 1, note, 100);
   ```

3. Eine MidiEvent-Instanz für die Message erzeugen.

   ```
   MidiEvent noteOn = new MidiEvent(a, 1);
   ```

4. Das Event zum Track hinzufügen.

   ```
   track.add(noteOn);
   ```

### Erstellen wir eine statische Hilfsmethode, die eine Message erzeugt und ein MidiEvent zurückgibt

Der Event-»Augenblick« – d.h., WANN diese Message über die Bühne gehen soll.

Die vier Argumente für die Message.

```
public static MidiEvent eventErzeugen(int comd, int chan, int one, int two, int tick) {
  MidiEvent event = null;
  try {
    ShortMessage a = new ShortMessage();
    a.setMessage(comd, chan, one, two);
    event = new MidiEvent(a, tick);

  }catch(Exception e) { }
  return event;
}
```

Hilfe! Eine Methode mit fünf Parametern!

Erzeugen Sie die Message und das Event mit Hilfe der Methodenparameter.

Geben Sie das Event zurück (ein MidiEvent, bis oben hin beladen mit der Message).

# Beispiel: Wie Sie die neue statische Methode eventErzeugen() verwenden

Hier gibt es keinerlei Event-Handling oder Grafik, sondern nur eine Abfolge von 15 Tönen, die immer höher werden. An diesem Code sollen Sie einfach nur lernen, wie unsere neue Methode eventErzeugen() benutzt wird. Dank dieser Methode wird dann der Code für die nächsten beiden Versionen wesentlich kürzer und einfacher werden.

```
import javax.sound.midi.*;

public class MiniMusikPlayer1 {

  public static void main(String[] args) {

    try {

      Sequencer sequencer = MidiSystem.getSequencer();
      sequencer.open();

      Sequence seq = new Sequence(Sequence.PPQ, 4);
      Track track = seq.createTrack();

      for (int i = 5; i < 61; i+= 4) {
        track.add(eventErzeugen(144,1,i,100,i));
        track.add(eventErzeugen(128,1,i,100,i + 2));
      } // Ende der Schleife

      sequencer.setSequence(seq);
      sequencer.setTempoInBPM(220);
      sequencer.start();
      Thread.sleep(5000);
      sequencer.close();
    } catch (Exception ex) {ex.printStackTrace();}
  } // main schließen

  public static MidiEvent eventErzeugen(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
      ShortMessage a = new ShortMessage();
      a.setMessage(comd, chan, one, two);
      event = new MidiEvent(a, tick);

    }catch(Exception e) { }
    return event;
  }
} // Klasse schließen
```

Vergessen Sie nicht den Import.

Erzeugen (und öffnen) Sie einen Sequencer.

Erzeugen Sie eine Sequence und einen Track.

Erzeugen Sie eine Reihe von Events, damit die Töne immer höher werden (vom Klavierton 5 bis zum Klavierton 61).

Rufen Sie unsere neue Methode eventErzeugen() auf und erzeugen Sie die Message und das Event. Fügen Sie dann das Ergebnis (das von eventErzeugen() zurückgegebene MidiEvent) zum Track hinzu. Dies hier sind NOTE ON- und NOTE OFF-Paare (144 bzw. 128).

Setzen Sie es in Gang.

# Version 2: Registrierung und Empfang von ControllerEvents

Wir müssen auf ControllerEvents lauschen, also müssen wir das Listener-Interface implementieren.

```
import javax.sound.midi.*;
public class MiniMusikPlayer2 implements ControllerEventListener {

  public static void main(String[] args) {
    MiniMusikPlayer2 mini = new MiniMusikPlayer2();
    mini.los();
  }

  public void los() {

    try {
      Sequencer sequencer = MidiSystem.getSequencer();
      sequencer.open();

      int[] gewünschteEvents = {127};
      sequencer.addControllerEventListener(this, gewünschteEvents);

      Sequence seq = new Sequence(Sequence.PPQ, 4);
      Track track = seq.createTrack();

      for (int i = 5; i < 60; i+= 4) {
        track.add(eventErzeugen(144,1,i,100,i));

        track.add(eventErzeugen(176,1,127,0,i));

        track.add(eventErzeugen(128,1,i,100,i + 2));
      } // Ende der Schleife

      sequencer.setSequence(seq);
      sequencer.setTempoInBPM(220);
      sequencer.start();
      Thread.sleep(5000);
      sequencer.close();
    } catch (Exception ex) {ex.printStackTrace();}
  } // Methode schließen

  public void controlChange(ShortMessage event) {
    System.out.println("la");
  }

  public MidiEvent eventErzeugen(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
      ShortMessage a = new ShortMessage();
      a.setMessage(comd, chan, one, two);
      event = new MidiEvent(a, tick);
    } catch (Exception e) { }
    return event;
  } // Methode schließen
} // Klasse schließen
```

Wir registrieren uns beim Sequencer für Events. Dieser Event-Registrierungsmethode werden der Listener UND ein int-Array übergeben; Letzteres repräsentiert die Liste der ControllerEvents, die Sie haben möchten. Wir wollen nur ein einziges Event, die Nummer 127.

Und so kommen wir an die Schläge: Wir fügen unser EIGENES ControllerEvent ein (176 heißt, es ist vom Event-Typ ControllerEvent) mit einem Argument für die Event-Nummer 127. Dieses Event tut NICHTS! Wir fügen es NUR DESHALB ein, damit wir überhaupt bei jedem Ton ein Event erhalten können. Anders ausgedrückt: Sein einziger Zweck ist, dass etwas abgesetzt wird, auf das WIR lauschen können (auf NOTE ON/OFF-Events können wir, wie gesagt, nicht lauschen). Beachten Sie, dass wir dieses Event im GLEICHEN Augenblick eintreten lassen wie NOTE ON. Wenn also das NOTE ON-Event eintritt, erfahren wir das, weil UNSER Event zum gleichen Zeitpunkt abgesetzt wird.

Die Event-Handler-Methode (aus dem ControllerEventListener-Interface). Jedes Mal, wenn wir das Event erhalten, geben wir auf der Kommandozeile >>la<< aus.

Code, der sich von der vorhergehenden Version unterscheidet, ist grau unterlegt. (Und wir lassen dieses Mal auch nicht alles in main() ausführen.)

# Version 3: Grafik im Takt der Musik

Diese endgültige Version baut auf Version 2 auf – es werden bloß noch die GUI-Anteile hinzugefügt. Wir erstellen einen Frame, fügen ein Zeichen-Panel hinzu, und jedes Mal, wenn wir ein Event erhalten, zeichnen wir ein neues Rechteck und aktualisieren den Bildschirm. Die einzige andere Veränderung gegenüber Version 2 ist, dass die Töne nicht immer höher werden, sondern nach dem Zufallsprinzip generiert werden.

Die wichtigste Veränderung im Code (neben der Erstellung einer einfachen Benutzeroberfläche) ist, dass wir den ControllerEventListener vom Zeichen-Panel implementieren lassen statt vom Programm selbst. Wenn also das Zeichen-Panel (eine innere Klasse) das Event erhält, weiß es selbst, was es zu tun hat, und zeichnet das Rechteck.

Der vollständige Code für diese Version steht auf der nächsten Seite.

## Die innere Klasse für das Zeichen-Panel:

Das Zeichen-Panel ist ein Listener.

```
class MeinZeichenPanel extends JPanel implements ControllerEventListener {

  boolean msg = false;
```

Wir setzen einen Schalter auf »false«, und er wird erst dann auf »true« gesetzt, wenn wir ein Event erhalten.

```
  public void controlChange(ShortMessage event) {
    msg = true;
    repaint();
  }
```

Wir haben ein Event erhalten, deshalb setzen wir den Schalter auf »true« und rufen repaint() auf.

```
  public void paintComponent(Graphics g) {
    if (msg) {
```

Wir müssen einen Schalter verwenden, weil auch ANDERE Dinge ein repaint() auslösen könnten, und wir wollen ja NUR bei einem ControllerEvent zeichnen.

```
      Graphics2D g2 = (Graphics2D) g;

      int r = (int) (Math.random() * 250);
      int gr = (int) (Math.random() * 250);
      int b = (int) (Math.random() * 250);

      g.setColor(new Color(r,gr,b));

      int höhe = (int) ((Math.random() * 120) + 10);
      int breite = (int) ((Math.random() * 120) + 10);
      int x = (int) ((Math.random() * 40) + 10);
      int y = (int) ((Math.random() * 40) + 10);
      g.fillRect(x,y,breite,höhe);
      msg = false;

    } // Ende der if-Verzweigung
  } // Methode schließen
} // innere Klasse schließen
```

Der Rest ist Code, mit dem eine zufällige Farbe erzeugt und ein teilweise zufälliges Rechteck gezeichnet wird.

**Spitzen Sie Ihren Bleistift**

Hier ist der vollständige Code für Version 3. Er baut direkt auf Version 2 auf. Versuchen Sie selbst die Anmerkungen zu schreiben, ohne sich die vorhergehenden Seiten anzusehen.

```
import javax.sound.midi.*;
import javax.swing.*;
import java.awt.*;

public class MiniMusikPlayer3 {

  static JFrame f = new JFrame("Mein erstes Musikvideo");
  static MeinZeichenPanel ml;

  public static void main(String[] args) {
    MiniMusikPlayer3 mini = new MiniMusikPlayer3();
    mini.los();
  } // Methode schließen

  public void guiErstellen() {
    ml = new MeinZeichenPanel();
    f.setContentPane(ml);
    f.setBounds(30, 30, 300, 300);
    f.setVisible(true);
    f.setDefaultCloseOperation(J.Frame.EXIT_ON_CLOSE);
  } // Methode schließen

  public void los() {
    guiErstellen();

    try {

      Sequencer sequencer = MidiSystem.getSequencer();
      sequencer.open();
      sequencer.addControllerEventListener(ml, new int[] {127});
      Sequence seq = new Sequence(Sequence.PPQ, 4);
      Track track = seq.createTrack();

      int r = 0;
      for (int i = 0; i < 60; i+= 4) {

        r = (int) ((Math.random() * 50) + 1);
        track.add(eventErzeugen(144,1,r,100,i));
        track.add(eventErzeugen(176,1,127,0,i));
        track.add(eventErzeugen(128,1,r,100,i + 2));
      } // Ende der for-Schleife

      sequencer.setSequence(seq);
      sequencer.start();
      Thread.sleep(5000);
      sequencer.close();
      sequencer.setTempoInBPM(120);

    } catch (Exception ex) {
      ex.printStackTrace();
    }
  } // Methode schließen
```

```
  public MidiEvent eventErzeugen(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
      ShortMessage a = new ShortMessage();
      a.setMessage(comd, chan, one, two);
      event = new MidiEvent(a, tick);

    } catch (Exception e) { }
    return event;
  } // Methode schließen


  class MeinZeichenPanel extends JPanel implements ControllerEventListener {
    boolean msg = false;

    public void controlChange(ShortMessage event) {
      msg = true;
      repaint();
    }

    public void paintComponent(Graphics g) {
      if (msg) {

        Graphics2D g2 = (Graphics2D) g;

        int r = (int) (Math.random() * 250);
        int gr = (int) (Math.random() * 250);
        int b = (int) (Math.random() * 250);

        g.setColor(new Color(r, gr, b));

        int höhe = (int) ((Math.random() * 120) + 10);
        int breite = (int) ((Math.random() * 120) + 10);

        int x = (int) ((Math.random() * 40) + 10);
        int y = (int) ((Math.random() * 40) + 10);

        g.fillRect(x, y, höhe, breite);
        msg = false;

      } // Ende der if-Verzweigung

    } // Methode schließen
  } // innere Klasse schließen

} // Klasse schließen
```

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

**Heute Abend anwesend:**

**Jede der reizenden Persönlichkeiten aus diesem Kapitel könnte hier anzutreffen sein!**

**Die ganze GUI liegt in meiner Hand.** _____

**Jeder Event-Typ hat so eins.** _____

**Die Schlüsselmethode des Listener.** _____

**Diese Methode gibt dem JFrame seine Größe.** _____

**Sie schreiben Code in diese Methode, aber Sie rufen sie nie auf.** _____

**Wenn der Benutzer tatsächlich etwas macht, ist das ein _____.** _____

**Die meisten davon sind Event-Quellen.** _____

**Ich überbringe dem Listener Daten.** _____

**Eine addXxxListener()-Methode bedeutet, ein Objekt ist eine _____.** _____

**Wie sich ein Listener anmeldet.** _____

**Die Methode, in der der ganze Grafikcode steht.** _____

**Typischerweise bin ich an eine Instanz gebunden.** _____

**Das »g« in (Graphics g) ist eigentlich vom Typ _____.** _____

**Die Methode, die paintComponent() in Gang setzt.** _____

**Das Paket, in dem es am meisten swingt.** _____

# Spielen Sie Compiler

Die Java-Datei auf dieser Seite stellt eine vollständige Quelldatei dar. Ihr Job ist es, Compiler zu spielen und festzustellen, ob sich diese Datei kompilieren lässt. Falls nein - wie könnten Sie das in Ordnung bringen? Und falls ja - was käme heraus?

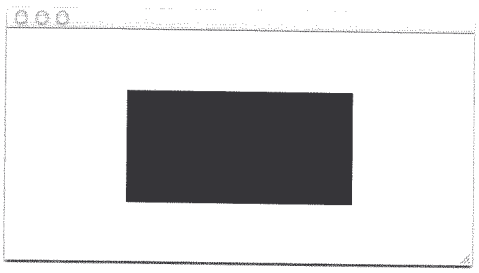
```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class InnererButton {

  JFrame frame;
  JButton b;

  public static void main(String [] args) {
    InnererButton gui = new InnererButton();
    gui.los();
  }

  public void los() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);

    b = new JButton("A");
    b.addActionListener();

    frame.getContentPane().add(
                BorderLayout.SOUTH, b);
    frame.setSize(200,100);
    frame.setVisible(true);
  }

  class BListener extends ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (b.getText().equals("A")) {
        b.setText("B");
      } else {
        b.setText("A");
      }
    }
  }
}
```

# Pool-Puzzle

Ihre ***Aufgabe*** ist es, die Codeschnipsel aus dem Pool zu nehmen und sie in die Lücken im Code einzusetzen. Jedes Schnipsel darf **mehr als einmal** verwendet werden, und Sie werden nicht alle Schnipsel benötigen. Ihr ***Ziel*** ist es, eine Klasse zu erstellen, die sich kompilieren und ausführen lässt und die hier gezeigte Ausgabe erzeugt.

**Ausgabe**

Das verblüffende, schrumpfende, blaue Rechteck. Dieses Programm liefert ein blaues Rechteck, das schrumpft und schrumpft und schrumpft ...



**Anmerkung: Jedes Schnipsel aus dem Pool kann mehr als einmal verwendet werden!**

```
import javax.swing.*;
import java.awt.*;
public class Animation {
  int x = 1;
  int y = 1;
  public static void main(String[] args) {
    Animation gui = new Animation();
    gui.los();
  }
  public void los() {
    JFrame _____ = new JFrame();
    frame.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE);
    ____________________________________________;
    _____.getContentPane().add(zP);
    ____________________________________;
    _____.setVisible(true);
    for (int i = 0; i < 124; _________) {
      ____________________;
      ____________________;
      try {
        Thread.sleep(50);
      } catch (Exception ex) {
      }
    }
  }
  class MeinZP extends JPanel {
    public void paintComponent(Graphics
                                    _________) {
      ____________________________________;
      ____________________________________;
      ____________________________________;
      ____________________________________;
    }
  }
}
```

g.fillRect(x,y,x-500,y-250)
g.fillRect(x,y,500-x*2,250-y*2)
g.fillRect(500-x*2,250-y*2,x,y)
g.fillRect(0,0,250,500)
g.fillRect(0,0,500,250)

x++
y++

i++
i++, y++
i++, y++, x++

g.setColor(blue)
g.setColor(white)
g.setColor(Color.blue)
g.setColor(Color.white)

g
z
frame
panel

zP.paint()
z.repaint()
zP.repaint()

Animation frame = new Animation()
MeinZP zP = new MeinZP()
ContentPane zP = new ContentPane()

zP.setSize(500,270)
frame.setSize(500,270)
panel.setSize(500,270)

# Lösungen zu den Übungen

## Wer bin ich?

| | |
|---|---|
| Die ganze GUI liegt in meiner Hand. | JFrame |
| Jeder Event-Typ hat so eins. | Listener-Interface |
| Die Schlüsselmethode des Listener. | actionPerformed() |
| Diese Methode gibt dem JFrame seine Größe. | setSize() |
| Sie schreiben Code in diese Methode, aber Sie rufen sie nie auf. | paintComponent() |
| Wenn der Benutzer tatsächlich etwas macht, ist das ein _____. | Event |
| Die meisten davon sind Event-Quellen. | Swing-Komponenten |
| Ich überbringe dem Listener Daten. | Event-Objekt |
| Eine addXxxListener()-Methode bedeutet, ein Objekt ist eine _____. | Event-Quelle |
| Wie sich ein Listener anmeldet. | addActionListener() |
| Die Methode, in der der ganze Grafikcode steht. | paintComponent() |
| Typischerweise bin ich an eine Instanz gebunden. | innere Klasse |
| Das »g« in (Graphics g) ist eigentlich vom Typ _____. | Graphics2D |
| Die Methode, die paintComponent() in Gang setzt. | repaint() |
| Das Paket, in dem es am meisten swingt. | javax.swing |

## Spielen Sie Compiler

Ist dieser Code erst einmal in Ordnung gebracht, erzeugt er eine GUI mit einem Button, dessen Beschriftung beim Anklicken zwischen A und B wechselt.

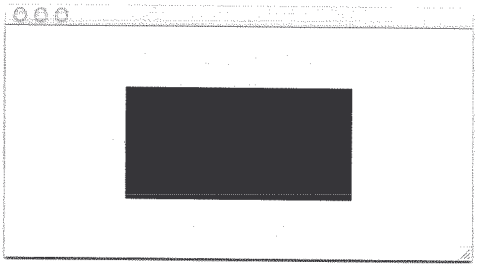
```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class InnererButton {

  JFrame frame;
  JButton b;

  public static void main(String[] args) {
    InnererButton gui = new InnererButton();
    gui.los();
  }

  public void los() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE);

    b = new JButton("A");
    b.addActionListener(new BListener());

    frame.getContentPane().add(
                    BorderLayout.SOUTH, b);
    frame.setSize(200, 100);
    frame.setVisible(true);
  }

  class BListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (b.getText().equals("A")) {
        b.setText("B");
      } else {
        b.setText("A");
      }
    }
  }
}
```

Die Methode addActionListener erhält als Argument eine Klasse, die das Interface ActionListener implementiert.

ActionListener ist ein Interface, und ein Interface wird implementiert, nicht erweitert.

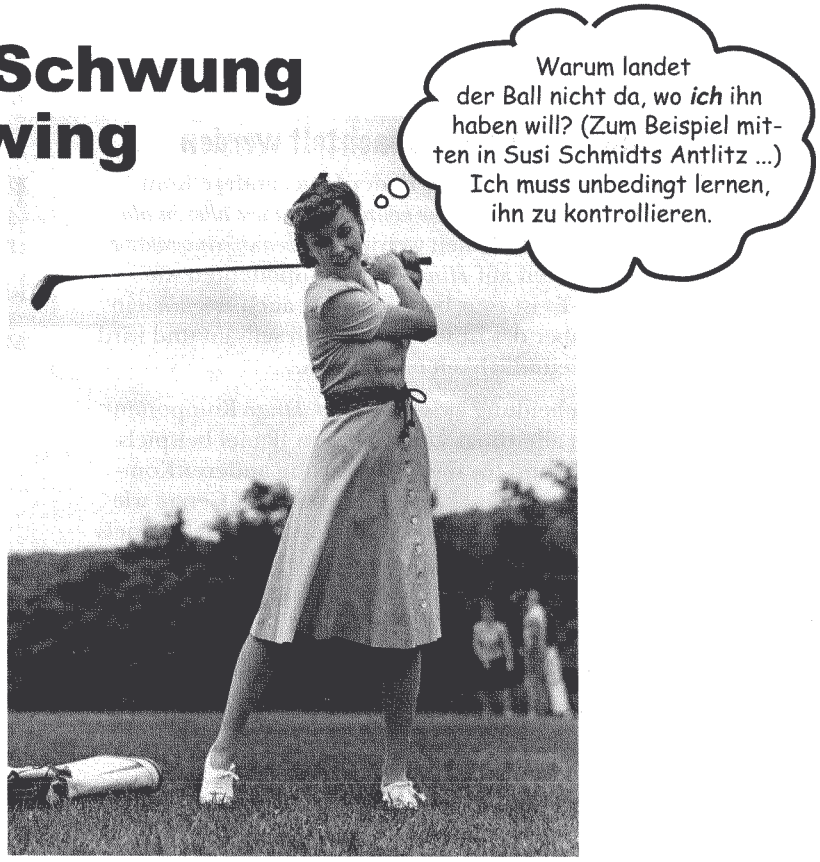
## Pool-Puzzle

Das verblüffende, schrumpfende, blaue Rechteck.



```
import javax.swing.*;
import java.awt.*;
public class Animation {
  int x = 1;
  int y = 1;
  public static void main(String[] args) {
    Animation gui = new Animation();
    gui.los();
  }
  public void los() {
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    MeinZP zP = new MeinZP();
    frame.getContentPane().add(zP);
    frame.setSize(500, 270);
    frame.setVisible(true);
    for (int i = 0; i < 124; i++, y++, x++) {
      x++;
      zP.repaint();
      try {
        Thread.sleep(50);
      } catch (Exception ex) {
      }
    }
  }
  class MeinZP extends JPanel {
    public void paintComponent(Graphics g) {
      g.setColor(Color.white);
      g.fillRect(0, 0, 500, 250);
      g.setColor(Color.blue);
      g.fillRect(x, y, 500 - x * 2, 250 - y * 2);
    }
  }
}
```

# Mehr Schwung mit Swing




**Swing ist einfach.** Wenn es Ihnen nicht so wichtig ist, wo die Dinge auf dem Bildschirm landen. Swing-Code *sieht einfach aus*, aber dann kompilieren Sie das Ganze, führen es aus, sehen es sich an und denken: »Hey, *das* sollte aber nicht *dort* stehen ...« Der Grund dafür, dass der Code so leicht zu *schreiben* ist, ist der **Layoutmanager** – aber genau der macht es auch so *schwer kontrollierbar*. Layoutmanager-Objekte steuern die Größe und Positionierung der Widgets in einer Java-GUI. Sie nehmen Ihnen eine Unmenge an Arbeit ab, doch die Ergebnisse werden Ihnen nicht immer gefallen. Sie möchten zwei gleich große Buttons, aber sie werden unterschiedlich groß. Sie wollen ein 8 cm langes Textfeld, doch es wird 25 cm lang. Oder 2 cm. Und es befindet sich *unter* der Beschriftung statt *daneben*. Mit ein bisschen Mühe bringen Sie Layoutmanager jedoch dazu, sich Ihrem Willen zu unterwerfen. In diesem Kapitel bringen wir unsere Swing-Künste in Schwung, lernen etwas über Layoutmanager und zudem auch noch mehr über Widgets. Wir erzeugen sie, zeigen sie an (dort, wo *wir* wollen!) und bringen sie in einem Programm zum Einsatz. Es sieht nicht gut aus für Susi.

# Swing-Komponenten

***Komponente*** ist der korrektere Begriff für das, was wir bisher als *Widget* bezeichnet haben. Die Dinger, die Sie in eine GUI setzen. *Das, was ein Benutzer sieht und mit dem er interagiert.* Textfelder, Buttons, scrollbare Listen, Radio-Buttons usw. sind Komponenten. Tatsächlich sind sie allesamt Unterklassen von `javax.swing.JComponent`.

**Ein Widget ist technisch gesehen eine Swing-Komponente. Fast alles, was Sie in eine GUI stecken können, erweitert javax.swing.JComponent.**

## Komponenten können ineinander verschachtelt werden

In Swing sind praktisch *alle* Komponenten dazu in der Lage, andere Komponenten zu enthalten. Anders ausgedrückt: *Sie können so gut wie alles in alles andere hineinstecken.* Aber in den meisten Fällen werden Sie *benutzerinteraktive* Komponenten wie Buttons oder Listen auf *Hintergrund*komponenten wie Frames und Panels unterbringen. Es ist zwar beispielsweise auch *möglich*, ein Panel in einen Button zu setzen, aber das ist schon ziemlich seltsam und wird Ihnen keinen Preis für Benutzerfreundlichkeit einbringen.

Außer bei JFrame ist diese Unterscheidung zwischen *interaktiven* Komponenten und *Hintergrund*komponenten allerdings künstlich. Ein JPanel beispielsweise wird in der Regel als Hintergrund für die Gruppierung anderer Komponenten verwendet, doch selbst ein JPanel kann interaktiv sein. Genau wie bei anderen Komponenten können Sie sich auch für die Events eines JPanels registrieren, einschließlich Mausklicks und Tastaturanschlägen.

### Die Erzeugung einer GUI in vier Schritten (Wiederholung)

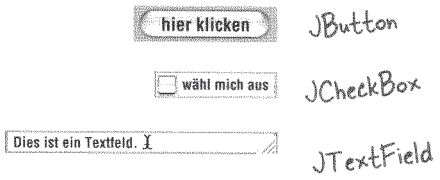
1. Erzeugen Sie ein Fenster (einen JFrame):
```
JFrame frame = new JFrame();
```

2. Erzeugen Sie eine Komponente (Button, Textfeld usw.):
```
JButton button = new JButton("klick mich");
```

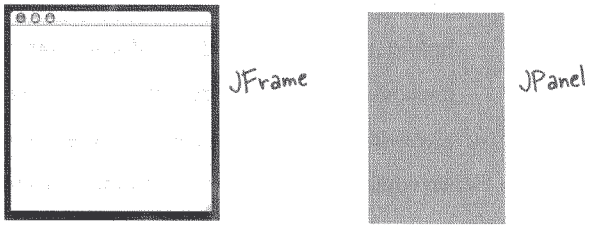
3. Fügen Sie die Komponente zum Frame hinzu:
```
frame.getContentPane().add(BorderLayout.EAST, button);
```

4. Zeigen Sie den Frame an
(geben Sie ihm eine Größe und machen Sie ihn sichtbar):
```
frame.setSize(300,300);
frame.setVisible(true);
```

### Setzen Sie interaktive Komponenten:

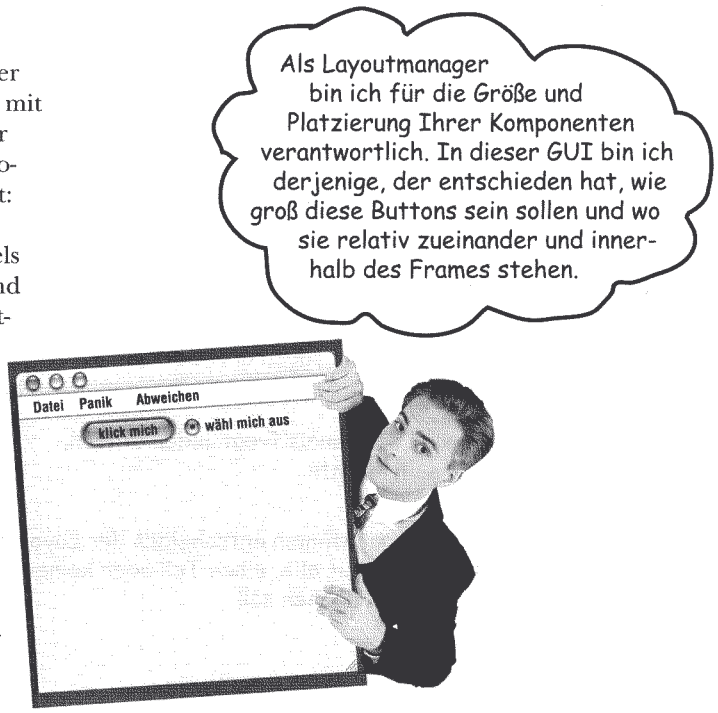



### ... in Hintergrundkomponenten:



# Layoutmanager

Ein Layoutmanager ist ein Java-Objekt, das mit einer bestimmten Komponente verknüpft ist, fast immer mit einer *Hintergrund*komponente. Der Layoutmanager kontrolliert die Komponenten *innerhalb* der Komponente, mit der er verknüpft ist. Anders ausgedrückt: Wenn ein Frame ein Panel enthält und das Panel einen Button, steuert der Layoutmanager des Panels die Größe und die Platzierung des Buttons, während der Layoutmanager des Frames die Größe und Platzierung des Panels steuert. Der Button hingegen braucht keinen Layoutmanager, weil er keine anderen Komponenten enthält.



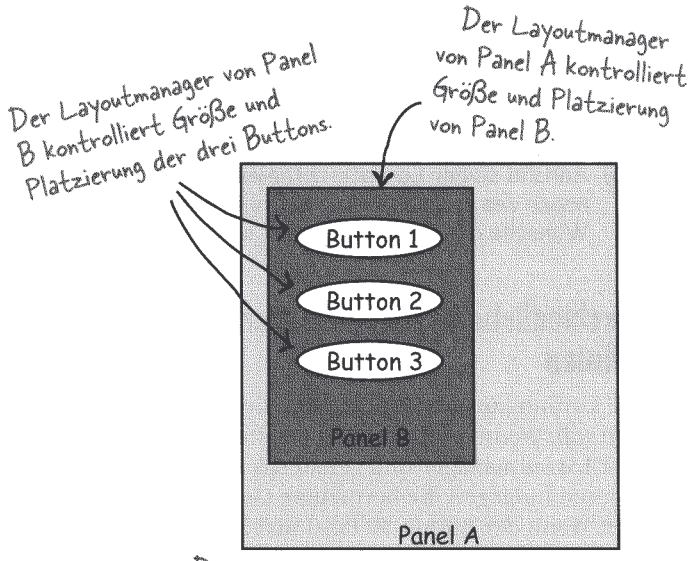

Wenn ein Panel fünf Dinge enthält, wird die Größe und Platzierung der fünf Dinge im Panel vom Layoutmanager des Panels kontrolliert, selbst wenn diese fünf Dinge ihre eigenen Layoutmanager haben. Enthalten diese fünf Dinge wiederum *andere* Dinge, werden diese *anderen* Dinge vom Layoutmanager desjenigen Elements positioniert, das sie enthält.

Wenn wir »enthält« sagen, meinen wir eigentlich, dass die Dinge mit Hilfe einer *add*-Methode hinzugefügt wurden – wie z.B. in »ein Panel enthält einen Button«, weil der Button zum Panel etwa wie folgt hinzugefügt wurde:

```
meinPanel.add(button);
```

Layoutmanager sind in unterschiedlichen Ausführungen erhältlich, und jede Hintergrundkomponente kann ihren eigenen Layoutmanager haben. Layoutmanager haben jeweils ihr eigenes Regelwerk, an das sie sich bei der Erstellung eines Layouts halten. So könnte beispielsweise ein bestimmter Layoutmanager darauf bestehen, dass alle Komponenten in einem Panel die gleiche Größe haben und in einem Raster angeordnet sind, während ein anderer Layoutmanager jeder Komponente die freie Wahl ihrer Größe lässt, aber alles vertikal übereinander stapelt. Hier ein Beispiel für geschachtelte Layouts:

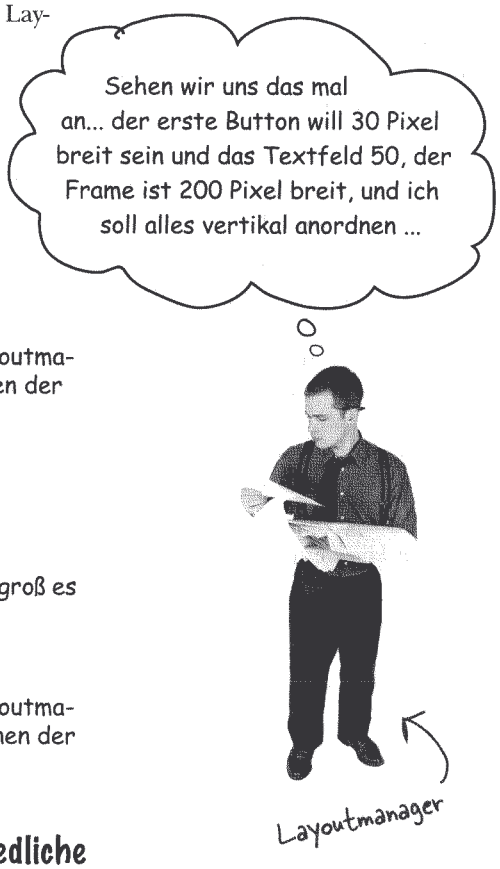
```
JPanel panelA = new JPanel();
JPanel panelB = new JPanel();
panelB.add(new JButton("Button 1"));
panelB.add(new JButton("Button 2"));
panelB.add(new JButton("Button 3"));
panelA.add(panelB);
```




Der Layoutmanager von Panel A kann NICHT über die drei Buttons bestimmen. Die Kontrollhierarchie reicht nur eine Stufe weit – der Layoutmanager von Panel A kontrolliert lediglich die Dinge, die direkt zu Panel A hinzugefügt werden, aber nichts, was innerhalb dieser hinzugefügten Komponenten liegt.

# Wie entscheidet der Layoutmanager?

Unterschiedliche Layoutmanager haben unterschiedliche Richtlinien für die Anordnung von Komponenten (z.B. Anordnung in einem Raster, Vorschreiben der gleichen Größe für alle, vertikale Stapelung usw.), aber die angeordneten Komponenten haben zumindest ein *kleines* Wörtchen dabei mitzureden. Im Allgemeinen wird beim Layout einer Hintergrundkomponente etwa wie folgt verfahren:




### Ein Layout-Szenario:

1. Sie erzeugen ein Panel und fügen drei Buttons hinzu.
2. Der Layoutmanager des Panels fragt jeden Button, wie groß er gern sein möchte.
3. Anhand seiner Layoutrichtlinien entscheidet der Layoutmanager des Panels, ob er auf alle, einen Teil oder keinen der Wünsche der Buttons eingehen soll.
4. Sie fügen das Panel zu einem Frame hinzu.
5. Der Layoutmanager des Frames fragt das Panel, wie groß es gern sein möchte.
6. Anhand seiner Layoutrichtlinien entscheidet der Layoutmanager des Frames, ob er auf alle, einen Teil oder keinen der Wünsche des Panels eingehen soll.

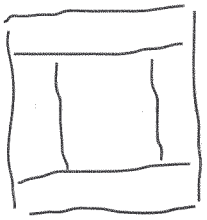
## Unterschiedliche Layoutmanager haben unterschiedliche Richtlinien

Manche Layoutmanager respektieren die Größe, die eine Komponente haben will. Wenn der Button 30 Pixel mal 50 Pixel groß sein will, weist der Layoutmanager ihm genau das zu. Andere Layoutmanager respektieren hingegen die bevorzugte Größe der Komponente nur teilweise. Wenn der Button 30 Pixel mal 50 Pixel groß sein will, sind es 30 Pixel mal der jeweiligen Breite des *Hintergrundpanels* von diesem Button. Wieder andere Layoutmanager respektieren nur die Wünsche der *größten* Komponente im Layout, und die restlichen Komponenten in dem Panel werden alle gleich groß gemacht. Manchmal kann die Arbeit der Layoutmanager sehr komplex sein, aber in den meisten Fällen kann man sich – wenn man die Richtlinien des Layoutmanagers erst einmal kennt – ausrechnen, was er tun wird.

# Die drei wichtigsten Layoutmanager: Border, Flow und Box

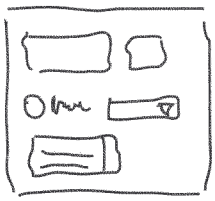
## BorderLayout

Ein BorderLayout-Manager teilt eine Hintergrundkomponente in fünf Bereiche. Zu einem Hintergrund, der von einem BorderLayout-Manager kontrolliert wird, können Sie nur *eine* Komponente pro Bereich hinzufügen. Von diesem Manager angeordnete Komponenten erhalten normalerweise nicht ihre bevorzugte Größe. **BorderLayout ist der Standard-Layoutmanager für Frames!**



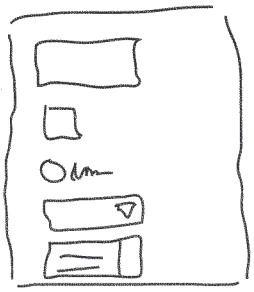

## FlowLayout

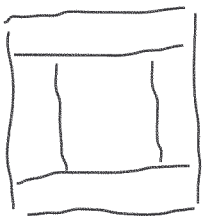
Ein FlowLayout-Manager arbeitet ähnlich wie eine Textverarbeitung, aber mit Komponenten statt Wörtern. Jede Komponente hat genau die Größe, die sie haben will, und die Komponenten werden von links nach rechts in genau der Reihenfolge angeordnet, in der sie hinzugefügt wurden – mit eingeschalteten »Zeilenumbruch«. Wenn also eine Komponente horizontal nicht mehr passt, wird sie in die nächste »Zeile« geschoben. **FlowLayout ist der Standard-Layoutmanager für Panels!**




## BoxLayout

Ein BoxLayout-Manager ähnelt dem FlowLayout insofern, als jede Komponente ihre gewünschte Größe erhält und die Komponenten in der Reihenfolge platziert werden, in der sie hinzugefügt werden. Aber – anders als beim FlowLayout – kann ein BoxLayout-Manager die Komponenten vertikal anordnen (oder horizontal, aber normalerweise haben wir es nur mit vertikal zu tun). Es ähnelt einem FlowLayout, aber an Stelle eines automatischen »Komponentenumbruchs« können Sie eine Art »Komponenten-Entertaste« drücken und die Komponenten so **zwingen**, eine neue Zeile zu beginnen.





## BorderLayout kümmert sich um fünf Bereiche:

## east, west, north, south und center

**Fügen wir einen Button zum east-Bereich – also im Osten – hinzu:**

```
import javax.swing.*;
import java.awt.*;

public class Button1 {

  public static void main(String[] args) {
    Button1 gui = new Button1();
    gui.los();
  }

  public void los() {
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JButton button = new JButton("klick mich");
    frame.getContentPane().add(BorderLayout.EAST, button);
    frame.setSize(200,200);
    frame.setVisible(true);
  }
}
```

BorderLayout gehört zum Paket java.awt.

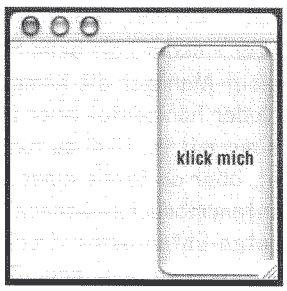
Hier wird der Bereich angegeben.

### Gehirntraining

Wie ist der BorderLayout-Manager auf diese Größe für den Button gekommen?

Welche Faktoren muss der Layoutmanager berücksichtigen?

Warum ist der Button nicht breiter oder höher?



**Sehen Sie, was passiert, wenn wir im Button mehr Zeichen einfügen:**

```
public void los() {
  JFrame frame = new JFrame();
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JButton button = new JButton("klick mich von Herzen");
  frame.getContentPane().add(BorderLayout.EAST, button);
  frame.setSize(200,200);
  frame.setVisible(true);
}
```

Wir haben nur den Text auf dem Button geändert.

Der Button bekommt seine bevorzugte Breite, aber nicht die gewünschte Höhe.

## Versuchen wir es mit einem Button im Norden:

```
public void los() {
  JFrame frame = new JFrame();
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JButton button = new JButton("innen hui, außen GUI ...");
  frame.getContentPane().add(BorderLayout.NORTH, button);
  frame.setSize(200,200);
  frame.setVisible(true);
}
```

Der Button ist so hoch, wie er sein möchte, aber so breit wie der Frame.

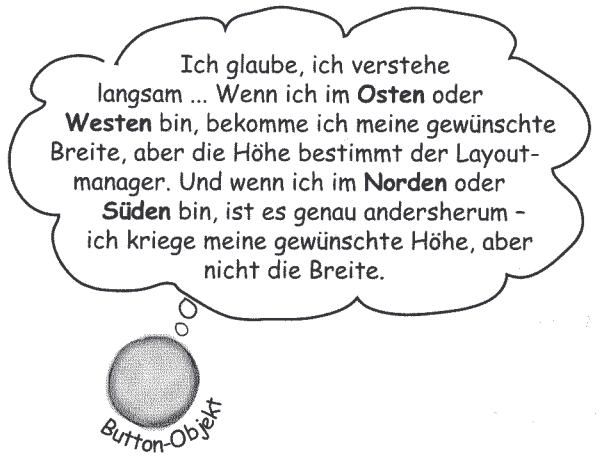
## Jetzt lassen wir den Button darum bitten, höher zu sein.

Wie machen wir das? Der Button ist schon so breit, wie er überhaupt sein kann - so breit wie der Rahmen. Aber wir können versuchen, ihn höher zu machen, indem wir ihm eine größere Schrift geben.

```
public void los() {
  JFrame frame = new JFrame();
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JButton button = new JButton("Klick hier!");
  Font bigFont = new Font("serif", Font.BOLD, 28);
  button.setFont(bigFont);
  frame.getContentPane().add(BorderLayout.NORTH, button);
  frame.setSize(200,200);
  frame.setVisible(true);
}
```

Ein größerer Font wird den Frame zwingen, mehr Platz für die Höhe des Buttons vorzusehen.

Die Breite bleibt gleich, aber der Button ist jetzt größer. Der nördliche Bereich wurde ausgedehnt, damit der Button in seiner neuen bevorzugten Höhe untergebracht werden kann.

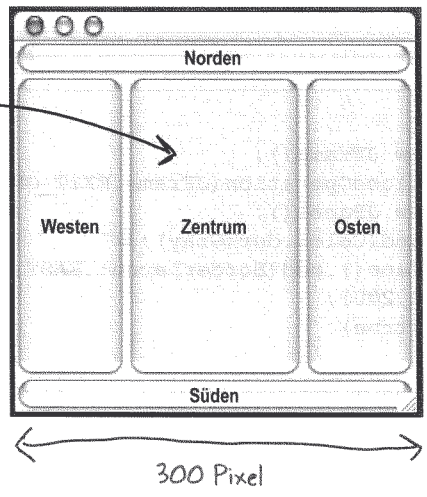



# Aber was passiert im zentralen Bereich?

## Der zentrale Bereich bekommt das, was übrig bleibt!

**(außer in einem Spezialfall, den wir uns später noch ansehen)**

```
public void los() {
  JFrame frame = new JFrame();
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  JButton east = new JButton("Osten");
  JButton west = new JButton("Westen");
  JButton north = new JButton("Norden");
  JButton south = new JButton("Süden");
  JButton center = new JButton("Zentrum");

  frame.getContentPane().add(BorderLayout.EAST, east);
  frame.getContentPane().add(BorderLayout.WEST, west);
  frame.getContentPane().add(BorderLayout.NORTH, north);
  frame.getContentPane().add(BorderLayout.SOUTH, south);
  frame.getContentPane().add(BorderLayout.CENTER, center);

  frame.setSize(300,300);
  frame.setVisible(true);
}
```

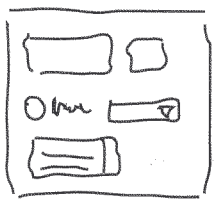



Komponenten im Zentrum kriegen das, was an Platz übrig ist, basierend auf den Maßen des Frames (hier 300 x 300).

Komponenten im Osten und Westen bekommen ihre bevorzugte Breite.

Komponenten im Norden und Süden bekommen ihre bevorzugte Höhe.

Wenn Sie etwas in den Norden oder Süden setzen, erstreckt es sich quer über den Frame, daher sind die Dinge im Osten und Westen nicht so hoch, als wenn der Nord- und der Südbereich leer wären.



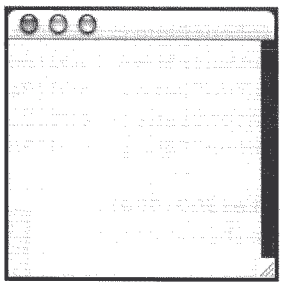
# Im FlowLayout fließen die Komponenten ähnlich wie ein Text:

## von links nach rechts, von oben nach unten, in der Reihenfolge, in der sie hinzugefügt wurden

### Fügen wir ein Panel im östlichen Bereich hinzu:

Der Standard-Layoutmanager für ein JPanel ist FlowLayout. Wenn wir ein Panel zu einem Frame hinzufügen, werden Größe und Platzierung des Panels wie gehabt vom BorderLayout-Manager gesteuert. Aber alles, was *innerhalb des Panels* liegt (mit anderen Worten: alle Komponenten, die durch den Aufruf von `panel.add(eineKomponente)` zum Panel hinzugefügt wurden), wird vom FlowLayout-Manager des Panels gesteuert.

Da in dem Panel nicht viel steht, beansprucht es nur eine geringe Breite im east-Bereich.



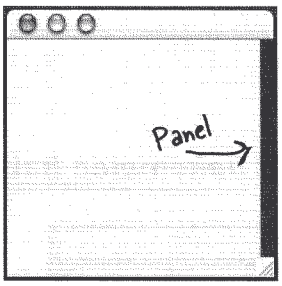
```
import javax.swing.*;
import java.awt.*;

public class Panel1 {

  public static void main (String[] args) {
    Panel1 gui = new Panel1();
    gui.los();
  }

  public void los() {
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel panel = new JPanel();
    panel.setBackground(Color.darkGray);
    frame.getContentPane().add(BorderLayout.EAST, panel);
    frame.setSize(200,200);
    frame.setVisible(true);
  }
}
```

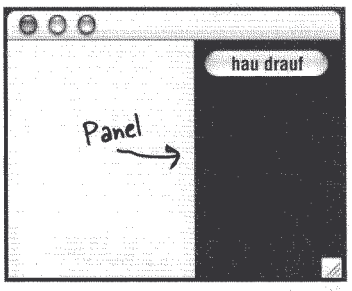
Machen wir das Panel grau, damit wir sehen können, wo innerhalb des Frames es sitzt.

## Fügen wir einen Button zum Panel hinzu

```
public void los() {
  JFrame frame = new JFrame();
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JPanel panel = new JPanel();
  panel.setBackground(Color.darkGray);

  JButton button = new JButton("hau drauf");

  panel.add(button);
  frame.getContentPane().add(BorderLayout.EAST, panel);

  frame.setSize(250,200);
  frame.setVisible(true);
}
```

Fügen Sie den Button zum Panel hinzu und das Panel zum Frame. Der LayoutManager des Panels (Flow) kontrolliert den Button, und der Layoutmanager des Frames (Border) kontrolliert das Panel.




Das Panel ist breiter geworden!

Und der Button hat seine bevorzugte Größe in beiden Dimensionen erhalten, weil das Panel mit FlowLayout arbeitet und der Button Teil des Panels ist (nicht des Frames).

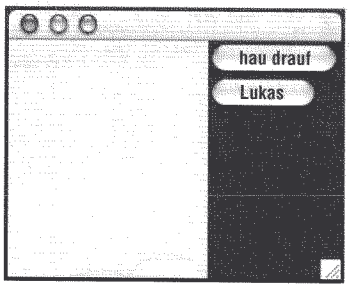
## Was passiert, wenn wir ZWEI Buttons zum Panel hinzufügen?

```
public void los() {
  JFrame frame = new JFrame();
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JPanel panel = new JPanel();
  panel.setBackground(Color.darkGray);

  JButton button = new JButton("hau drauf");
  JButton button2 = new JButton("Lukas");

  panel.add(button);
  panel.add(button2);

  frame.getContentPane().add(BorderLayout.EAST, panel);
  frame.setSize(250,200);
  frame.setVisible(true);
}
```

ZWEI Buttons machen

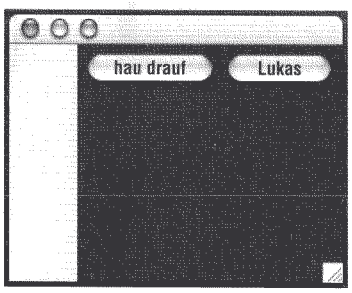
BEIDE zum Panel hinzufügen

**Was wir *wollten*:**




Wir wollten die Buttons übereinander gestapelt haben.

**Was wir *bekommen haben*:**



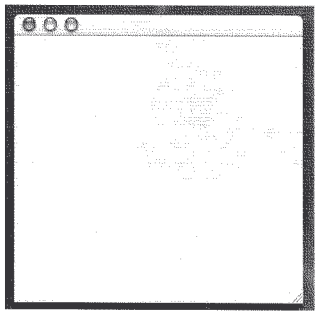

Das Panel hat sich verbreitert, so dass beide Buttons nebeneinander passen.

Beachten Sie, dass der >>Lukas<<-Button kleiner ist als der >>hau drauf<<-Button ... So funktioniert FlowLayout. Der Button bekommt genau das, was er braucht (und nicht mehr).

### Spitzen Sie Ihren Bleistift

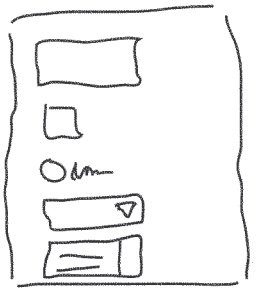
Wie sähe die GUI aus, wenn der obige Code in die unten stehende Fassung abgewandelt würde?

```
JButton button = new JButton("hau drauf");
JButton button2 = new JButton("Lukas");
JButton button3 = new JButton("wie jetzt?");
panel.add(button);
panel.add(button2);
panel.add(button3);
```



Zeichnen Sie die GUI, wie sie Ihrer Ansicht nach aussehen müsste, wenn man den links stehenden Code ausführt.

(Und dann probieren Sie es aus!)



# BoxLayout, zu Hilfe!

## Es sorgt für die Stapelung der Komponenten, auch wenn genug Platz ist, sie nebeneinander zu setzen.

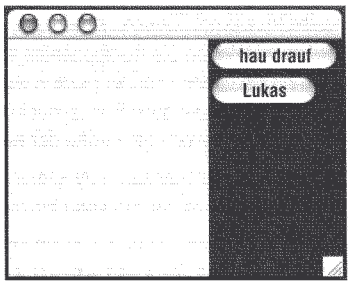
**Anders als FlowLayout kann BoxLayout einen »Zeilenumbruch« erzwingen, damit die Komponenten in der nächsten Zeile stehen – selbst wenn genug Platz wäre, um sie nebeneinander zu setzen.**

Aber dafür müssen Sie den Layoutmanager des Panels vom Standard-FlowLayout auf BoxLayout umstellen.

```
public void los() {
  JFrame frame = new JFrame();
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JPanel panel = new JPanel();
  panel.setBackground(Color.darkGray);

  panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));

  JButton button = new JButton("hau drauf");
  JButton button2 = new JButton("Lukas");
  panel.add(button);
  panel.add(button2);
  frame.getContentPane().add(BorderLayout.EAST, panel);
  frame.setSize(250,200);
  frame.setVisible(true);
}
```

Erzeugen Sie für den Layoutmanager eine neue Instanz von BoxLayout.

Der BoxLayout-Konstruktor muss wissen, für welche Komponente er das Layout macht (d.h. das Panel) und welche Achse er verwenden soll (wir nehmen die y-Achse, weil wir eine vertikale Stapelung wollen).




Beachten Sie, dass das Panel jetzt wieder schmaler ist, weil es nicht beide Buttons nebeneinander unterbringen muss. Das Panel hat daher dem Frame mitgeteilt, dass es nicht mehr Platz braucht, als zum Unterbringen des größten Buttons (»hau drauf«) nötig ist.

## Es gibt keine Dummen Fragen

**F: Wie kommt es, dass man zu einem Frame nicht direkt etwas hinzufügen kann – so wie zu einem Panel?**

A: Ein JFrame ist eine Besonderheit, denn hier, an der Grenze unseres Systems, kommt der Moment der Wahrheit, wenn etwas auf dem Bildschirm erscheinen soll. All Ihre Swing-Komponenten sind reines Java, ein JFrame dagegen muss sich mit dem darunter liegenden Betriebssystem verbinden, um Zugriff auf die Anzeige zu bekommen. Stellen Sie sich die »content pane« als eine 100% reine Java-Schicht vor, die *oben* auf dem JFrame sitzt. Oder stellen Sie sich vor, JFrame wäre der Fensterrahmen und die Content-Pane die ... Fensterscheibe (das englische Wort *pane* bedeutet nämlich nichts anderes als »Scheibe«). Und Sie können die Content-Pane sogar mit Hilfe der folgenden Anweisung gegen Ihr eigenes JPanel *austauschen* und so das JPanel zur Content-Pane des Frames machen:

```
meinFrame.setContentPane(meinPanel); *
```

**F: Kann ich den Layoutmanager des Frames austauschen? Was ist, wenn mein Frame Flow- statt BorderLayout verwenden soll?**

A: Der einfachste Weg dafür ist, ein Panel zu erstellen, die Oberfläche im Panel auf die gewünschte Weise aufzubauen und dann das Panel zur Content-Pane des Frames zu machen, wie es der Code in der vorhergehenden Antwort macht (statt die Standard-Content-Pane zu benutzen).

**F: Was ist, wenn ich eine andere bevorzugte Größe möchte? Gibt es eine Methode setSize() für Komponenten?**

A: Ja, es gibt setSize(), aber die Layoutmanager ignorieren das. Es gibt einen Unterschied zwischen der *bevorzugten Größe* der Komponente und der Größe, in der *Sie* sie gern hätten. Die bevorzugte Größe basiert darauf, welche Größe die Komponente tatsächlich *braucht* (das entscheidet die Komponente selbst). Der Layoutmanager ruft die Methode getPreferredSize() der Komponente auf, und *diese* Methode schert sich nicht darum, ob Sie vorher setSize() auf der Komponente aufgerufen haben.

**F: Kann ich Dinge nicht einfach dahin setzen, wo ich sie haben will? Kann ich die Layoutmanager ausschalten?**

A: Ja! Sie können Komponente um Komponente `setLayout(null)` aufrufen – und dann ist es Ihre Sache, den Code für die genauen Bildschirmplatzierungen und -abmessungen selbst zu schreiben. Letztendlich ist es aber fast immer einfacher, die Layoutmanager zu verwenden.

* Wie wir bereits auf S. 354 angedeutet haben, dürfen Sie seit Java 5.0 auch meinFrame.add(button) schreiben. Das erspart Ihnen viel Tipparbeit, und Ihr Code wird übersichtlicher. Hinter den Kulissen fügt Swing jedoch Ihren Button wie gehabt zur Content-Pane hinzu, nicht direkt zum JFrame.

## Punkt für Punkt

- Layoutmanager steuern die Größe und Platzierung von Komponenten, die mit anderen Komponenten verschachtelt sind.
- Wenn Sie eine Komponente zu einer anderen Komponente (die manchmal auch als *Hintergrund*komponente bezeichnet wird, aber das ist keine fachlich exakte Unterscheidung) hinzufügen, wird die hinzugefügte Komponente durch den Layoutmanager der *Hintergrund*komponente kontrolliert.
- Ein Layoutmanager fragt Komponenten nach ihrer bevorzugten Größe, bevor er eine Entscheidung über das Layout trifft. Abhängig von den Richtlinien des Layoutmanagers wird er die Wünsche der Komponente dann alle, zum Teil oder gar nicht berücksichtigen.
- Beim BorderLayout-Manager können Sie eine Komponente in einem von fünf Bereichen unterbringen. Beim Hinzufügen der Komponente müssen Sie den Bereich mit der folgenden Syntax angeben:

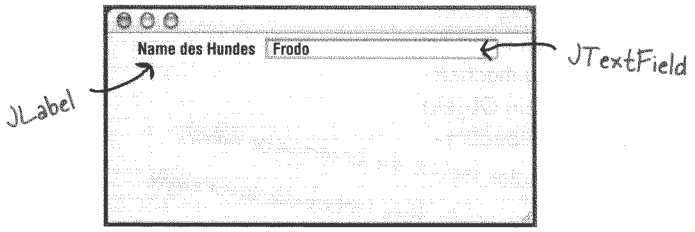
  ```
  add(BorderLayout.EAST, panel);
  ```

- Beim BorderLayout erhalten Komponenten im Norden und Süden ihre bevorzugte Höhe, aber nicht die bevorzugte Breite. Komponenten im Osten und im Westen erhalten die bevorzugte Breite, aber nicht die bevorzugte Höhe. Die Komponente in der Mitte bekommt das, was übrigt bleibt (es sei denn, Sie verwenden `pack()`).
- Die Methode pack() ist eine Art Schrumpffolie für die Komponenten; sie verwendet die volle bevorzugte Größe der center-Komponente als Basis und ermittelt dann die Größe des Frames, indem sie die Inhalte der anderen Bereiche um den center-Bereich herum aufbaut.
- In einem FlowLayout werden Komponenten von links nach rechts und von oben nach unten angeordnet, und zwar in der Reihenfolge, in der sie hinzugefügt werden. Ein Zeilenumbruch erfolgt nur dann, wenn die Komponenten horizontal nicht nebeneinander passen.
- Im FlowLayout erhalten Komponenten in beiden Dimensionen ihre bevorzugte Größe.
- Im BoxLayout können Sie Komponenten vertikal anordnen, selbst wenn sie nebeneinander passen würden. Wie FlowLayout verwendet auch BoxLayout in beiden Dimensionen die bevorzugte Größe der Komponente.
- BorderLayout ist der Standard-Layoutmanager bei einem Frame; FlowLayout ist der Standard bei einem Panel.
- Wenn ein Panel etwas anderes verwenden soll als FlowLayout, müssen Sie `setLayout()` auf dem Panel aufrufen.

# Spielen mit Swing-Komponenten

Die Grundlagen zum Arbeiten mit Layoutmanagern beherrschen Sie jetzt; lassen Sie uns nun ein paar der häufigsten Komponenten ausprobieren: ein Textfeld, einen scrollbaren Textbereich, ein Kontrollkästchen und eine Liste. Allerdings wollen wir Sie hier nicht mit der gesamten API quälen, sondern zeigen Ihnen nur ein paar Highlights, damit Sie schon einmal loslegen können.

## JTextField




### Konstruktoren

```
JTextField field = new JTextField(20);
```

20 bedeutet 20 Spalten, nicht 20 Pixel. Damit wird die bevorzugte Breite des Textfelds festgelegt.

```
JTextField field = new JTextField("Ihr Name");
```

### Verwendung

1. Text herausholen

```
System.out.println(field.getText());
```

2. Text hineinsetzen

```
field.setText("irgendwas");
field.setText("");
```

Damit wird der Feldinhalt gelöscht.

3. Ein ActionEvent erhalten, wenn der Benutzer auf Return oder Enter drückt

Sie können sich auch für Tastatur-Events registrieren, wenn Sie es tatsächlich jedes Mal wissen wollen, wenn der Benutzer auf eine Taste drückt.

```
field.addActionListener(meinActionListener);
```

4. Den Text im Feld auswählen/hervorheben

```
field.selectAll();
```

5. Den Cursor zurück in das Feld setzen (damit der Benutzer sofort anfangen kann zu tippen)

```
field.requestFocus();
```

## JTextArea

Anders als ein JTextField kann eine JTextArea (ein Textbereich) mehr als eine Zeile Text enthalten. Ihre Erzeugung erfordert ein wenig Konfigurationsaufwand, weil sie nicht von vornherein mit Scrollbalken oder Zeilenumbruch ausgestattet ist. Um eine JTextArea scrollfähig zu machen, müssen Sie sie in eine Scroll-Pane setzen. Eine Scroll-Pane ist ein Objekt mit einer besonderen Vorliebe fürs Scrollen, das sich der Scrollbedürfnisse des Textbereichs annehmen wird.

### Konstruktor

```
JTextArea text = new JTextArea(10,20);
```

10 bedeutet 10 Zeilen (setzt die bevorzugte Höhe).

20 bedeutet 20 Spalten (setzt die bevorzugte Breite).

### Verwendung

1. Nur ein vertikaler, kein horizontaler Scrollbalken

```
JScrollPane scroller = new JScrollPane(text);
text.setLineWrap(true);

scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(scroller);
```

Erzeugen Sie eine JScrollPane und übergeben Sie ihr den Textbereich, für den sie scrollen soll.

Schalten Sie den Zeilenumbruch ein.

Sagen Sie der Scroll-Pane, sie soll nur einen vertikalen Scrollbalken benutzen.

Wichtig!! Sie übergeben den Textbereich der Scroll-Pane (über den Scroll-Pane-Konstruktor) und fügen dann die Scroll-Pane zum Panel hinzu. Sie fügen also den Textbereich nicht direkt zum Panel hinzu!

2. Den darin stehenden Text ersetzen

```
text.setText("Nicht jeder Verlor'ne wandert");
```

3. Etwas an den schon vorhandenen Text anhängen

```
text.append("Button angeklickt");
```

4. Den Text im Feld auswählen/hervorheben

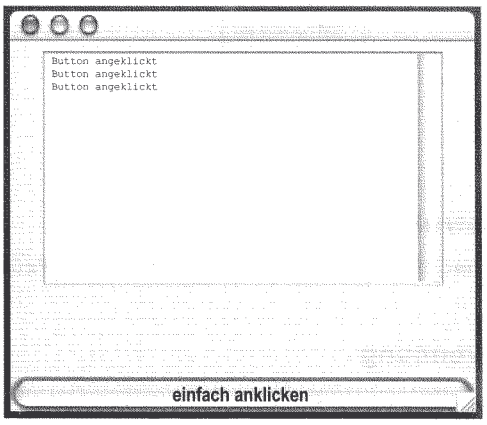
```
text.selectAll();
```

5. Den Cursor zurück in das Feld setzen (damit der Benutzer sofort anfangen kann zu tippen)

```
text.requestFocus();
```

## Beispiel für JTextArea

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

public class Textbereich1 implements ActionListener {

  JTextArea text;

  public static void main(String[] args) {
    Textbereich1 gui = new Textbereich1();
    gui.los();
  }

  public void los() {
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel panel = new JPanel();
    JButton button = new JButton("einfach anklicken");
    button.addActionListener(this);
    text = new JTextArea(10,20);
    text.setLineWrap(true);

    JScrollPane scroller = new JScrollPane(text);
    scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
    scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

    panel.add(scroller);

    frame.getContentPane().add(BorderLayout.CENTER, panel);
    frame.getContentPane().add(BorderLayout.SOUTH, button);

    frame.setSize(350,300);
    frame.setVisible(true);
  }

  public void actionPerformed(ActionEvent ev) {
    text.append("Button angeklickt \n");
  }
}
```




Hier wird eine neue Zeile eingefügt, so dass bei jedem Klick auf den Button eine neue Zeile begonnen wird. Sonst würde alles wie hier ineinander laufen.

## JCheckBox

### Konstruktor

```
JCheckBox check = new JCheckBox("Früh krümmt sich");
```

### Verwendung

1. Auf ein Item-Event lauschen (wenn ein Häkchen gesetzt bzw. entfernt wird)

```
check.addItemListener(this);
```

2. Das Event behandeln (und herausfinden, ob die Checkbox ausgewählt ist oder nicht)

```
public void itemStateChanged(ItemEvent ev) {
  String jaOderNein = "nein";
  if (check.isSelected()) jaOderNein = "ja";
  System.out.println("Häkchen gesetzt: " + jaOderNein);
}
```

3. Auswahl oder Nichtauswahl im Code voreinstellen

```
check.setSelected(true);
check.setSelected(false);
```

## Es gibt keine Dummen Fragen

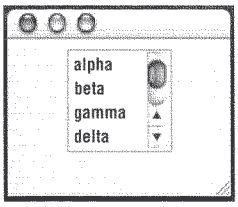
**F: Machen die Layoutmanager nicht mehr Ärger, als sie wert sind? Statt mich mit all diesen Problemen herumzuschlagen, könnte ich doch genauso gut einfach die Größe und die Koordinaten für alles selbst in den Code schreiben.**

A: Genau das gewünschte Layout von einem Layoutmanager zu erhalten ist manchmal tatsächlich nicht einfach. Aber überlegen Sie mal, was der Layoutmanager tatsächlich für Sie tut. Selbst die leicht erscheinende Aufgabe, nämlich zu berechnen, wo die Dinge auf dem Bildschirm stehen sollen, kann kompliziert sein. So kümmert sich beispielsweise der Layoutmanager darum, dass Ihre Komponenten sich nicht überlappen. Anders ausgedrückt, er weiß, wie man die Abstände zwischen Komponenten (und zum Rand des Frames) einhält. Sicher können Sie das auch selbst tun, aber was passiert, wenn Sie Komponenten sehr dicht packen wollen? Sie könnten sie natürlich von Hand genau richtig positionieren, aber das funktioniert dann nur auf *Ihrer* JVM.

Warum? Weil sich die Komponenten von Plattform zu Plattform etwas unterscheiden, vor allem wenn sie das native »Look-and-Feel« der darunter liegenden Plattform benutzen. Feine Unterschiede, z.B. bei der Abschrägung der Buttons, können dazu führen, dass Komponenten, die auf einer Plattform hübsch aufgereiht sind, auf einer anderen plötzlich aneinander gequetscht werden.

Und die wichtigste Aufgabe von Layoutmanagern haben wir noch gar nicht erwähnt. Denken Sie daran, was passiert, wenn der Benutzer die Größe des Fensters verändert! Oder wenn Ihre Oberfläche dynamisch ist, mit Komponenten, die kommen und gehen. Wenn Sie all das selbst verfolgen und die Komponenten jedes Mal neu anordnen müssten, falls sich die Größe oder der Inhalt einer Hintergrundkomponente verändern ... oh je!

## JList




Der JList-Konstruktor nimmt ein Array mit einem beliebigen Objekttyp entgegen. Das müssen keine Strings sein, aber in der Liste werden sie durch Strings repräsentiert.

### Konstruktor

```
String [] listEntries = {"alpha", "beta", "gamma", "delta",
                         "epsilon", "zeta", "eta", "theta"};
JList list = new JList(listEntries);
```

### Verwendung

Genau wie bei JTextArea – Sie machen eine JScrollPane (und übergeben ihr die Liste), und dann fügen Sie die Scroll-Pane (NICHT die Liste) zum Panel hinzu.

① Einen vertikalen Scrollbalken hinzufügen

```
JScrollPane scroller = new JScrollPane(list);
scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(scroller);
```

② Anzahl der Zeilen festlegen, die vor dem Scrollen angezeigt werden sollen

```
list.setVisibleRowCount(4);
```

③ Festlegen, dass der Benutzer nur EINEN Eintrag auf einmal auswählen darf

```
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

④ Für Listenauswahl-Events registrieren

```
list.addListSelectionListener(this);
```

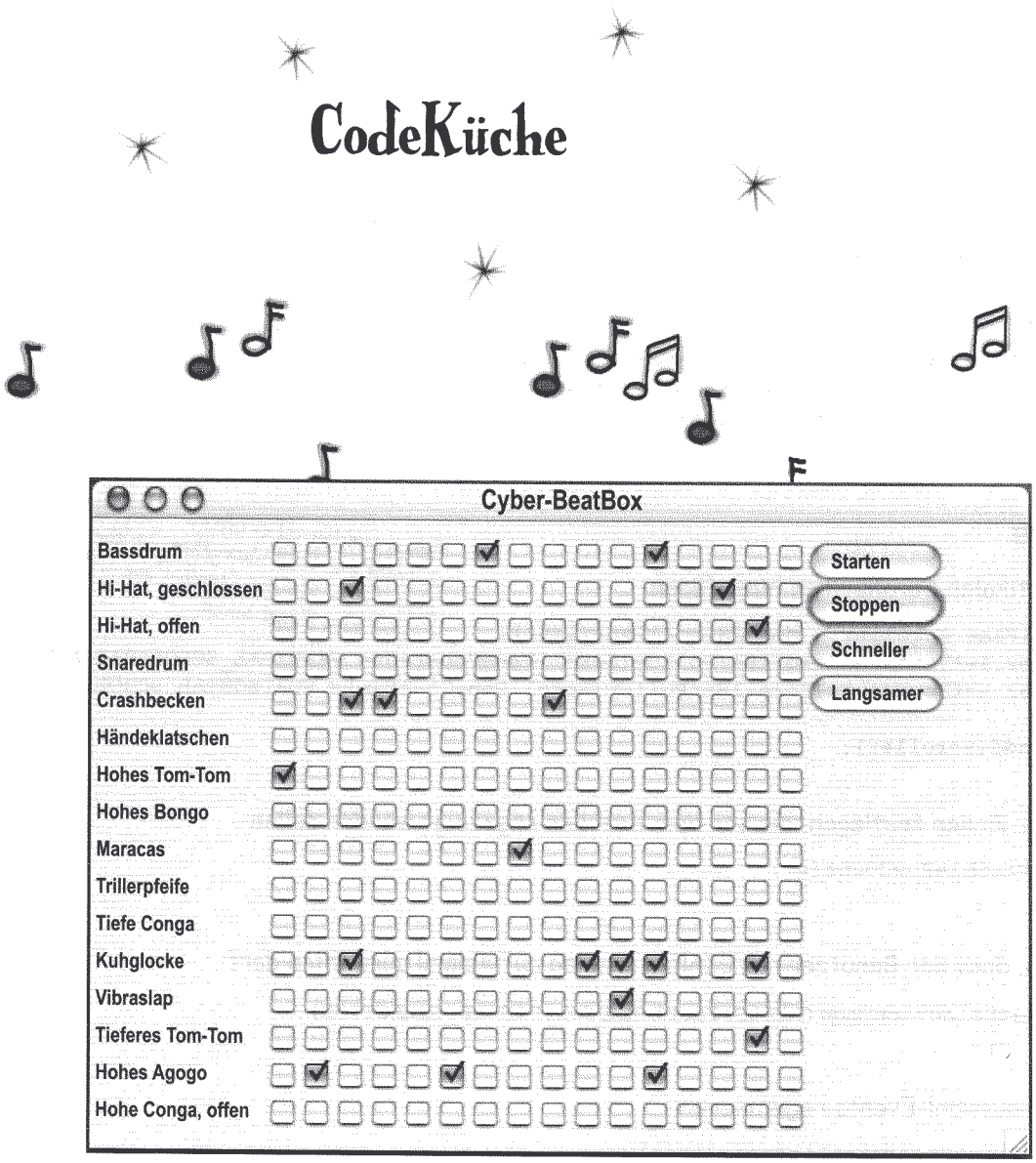
⑤ Events behandeln (herausfinden, welcher Listeneintrag ausgewählt wurde)

```
public void valueChanged(ListSelectionEvent lse) {
  if( !lse.getValueIsAdjusting()) {
    String selection = (String) list.getSelectedValue();
    System.out.println(selection);
  }
}
```

Wenn Sie diese if-Abfrage weglassen, bekommen Sie das Event ZWEIMAL.

getSelectedValue() gibt eigentlich ein Object zurück. Eine Liste kann nicht nur String-Objekte enthalten.

# CodeKüche




Dieser Teil ist optional. Wir bauen die komplette BeatBox, mit GUI und allem Pipapo. In dem Kapitel über das Speichern von Objekten werden wir dann noch lernen, wie man Drum-Patterns speichert und wiederherstellt. Und schließlich verwandeln wir die BeatBox in dem Kapitel über Netzwerke in einen voll funktionsfähigen Chatclient.

# Die BeatBox erstellen

Hier kommt der vollständige Code für diese Version der BeatBox – mit den Buttons für das Starten, Stoppen und Verändern des Tempos. Der Code ist vollständig abgedruckt, mit allen Anmerkungen, doch hier erst einmal ein Überblick:

1. Wir erstellen eine Benutzeroberfläche mit 256 Kontrollkästchen (vom Typ JCheckBox), die anfangs nicht ausgewählt sind, mit 16 Beschriftungen (vom Typ JLabel) für die Instrumentnamen und mit 4 Buttons.

2. Für jeden der vier Buttons registrieren wir einen ActionListener. Wir brauchen keine Listener für die einzelnen Kontrollkästchen, weil wir nicht versuchen, den Sound des Patterns dynamisch (d.h., sobald der Benutzer eine Checkbox auswählt) zu ändern. Stattdessen warten wir, bis der Benutzer auf den Button »Starten« drückt; dann gehen wir alle 256 Kontrollkästchen durch, stellen ihren Zustand fest und erzeugen einen MIDI-Track.

3. Wir erstellen das MIDI-System (das haben Sie ja schon einmal gemacht), indem wir uns einen Sequencer geben lassen, eine Sequence machen und einen Track erzeugen. Wir verwenden eine Sequencer-Methode, die neu in Java 5.0 ist: setLoopCount(). Mit Hilfe dieser Methode können Sie angeben, wie oft Sie eine Sequence wiederholen möchten. Außerdem benutzen wir den Tempofaktor der Sequence, um das Tempo nach oben oder unten anzupassen, und halten dieses neue Tempo, wenn wir von einer Iteration der Schleife zur nächsten übergehen.

4. Wenn der Benutzer auf »Starten« klickt, geht es richtig los. Die Event-Handling-Methode für den »Starten«-Button ruft die Methode trackErstellenUndStarten() auf. In dieser Methode gehen wir alle 256 Kontrollkästchen durch (reihenweise, d.h. ein einzelnes Instrument über alle 16 Schläge), um ihren Zustand abzufragen, und erstellen dann mit diesen Informationen einen MIDI-Track (mit der praktischen eventErzeugen()-Methode, die wir im letzten Kapitel benutzt haben). Wenn wir den Track erstellt haben, starten wir den Sequencer, der immer weiterspielt (weil wir eine Schleife für ihn programmiert haben) - so lange, bis der Benutzer auf »Stoppen« klickt.

```
import java.awt.*;
import javax.swing.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;


public class BeatBox {


  JPanel hauptPanel;
  ArrayList<JCheckBox> checkboxListe;
  Sequencer sequencer;
  Sequence sequence;
  Track track;
  JFrame derFrame;

  String[] instrumentNamen = {"Bassdrum", "Hi-Hat, geschlossen",
     "Hi-Hat, offen", "Snaredrum", "Crashbecken", "Händeklatschen",
     "Hohes Tom-Tom", "Hohes Bongo", "Maracas", "Trillerpfeife",
     "Tiefe Conga", "Kuhglocke", "Vibraslap", "Tieferes Tom-Tom",
     "Hohes Agogo", "Hohe Conga, offen"};
  int[] instrumente = {35,42,46,38,49,39,50,60,70,72,64,56,58,47,67,63};


  public static void main (String[] args) {
    new BeatBox().guiErstellen();
  }

  public void guiErstellen() {
    derFrame = new JFrame("Cyber-BeatBox");
    derFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    BorderLayout layout = new BorderLayout();
    JPanel hintergrund = new JPanel(layout);
    hintergrund.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

    checkboxListe = new ArrayList<JCheckBox>();
    Box buttonBox = new Box(BoxLayout.Y_AXIS);

    JButton start = new JButton("Starten");
    start.addActionListener(new MeinStartListener());
    buttonBox.add(start);

    JButton stopp = new JButton("Stoppen");
    stopp.addActionListener(new MeinStoppListener());
    buttonBox.add(stopp);

    JButton schneller = new JButton("Schneller");
    schneller.addActionListener(new MeinSchnellerListener());
    buttonBox.add(schneller);
```

Wir speichern die Kontrollkästchen in einer ArrayList.

Dies sind die Namen der Instrumente in Form eines String-Arrays, mit dem die GUI-Labels (für jede Reihe) erstellt werden.

Dies sind die eigentlichen Drum-»Tasten«. Der Drum-Kanal ähnelt einem Klavier, bei dem aber jede Klavier-»Taste« einem anderen Percussion-Instrument entspricht. So ist beispielsweise die Zahl 35 die Taste für die Basstrommel, 42 ist das geschlossene Hi-Hat usw.

Ein »leerer Rand« sorgt für einen Abstand zwischen den Rändern des Panels und den Komponenten. Aus rein ästhetischen Gründen.

Dies hier ist nichts Besonderes, einfach nur viel GUI-Code. Das meiste davon kennen Sie bereits.

```
  JButton langsamer = new JButton("Langsamer");
  langsamer.addActionListener(new MeinLangsamerListener());
  buttonBox.add(langsamer);

  Box namensBox = new Box(BoxLayout.Y_AXIS);
  for (int i = 0; i < 16; i++) {
    namensBox.add(new Label(instrumentNamen[i]));
  }

  hintergrund.add(BorderLayout.EAST, buttonBox);
  hintergrund.add(BorderLayout.WEST, namensBox);

  derFrame.getContentPane().add(hintergrund);

  GridLayout raster = new GridLayout(16,16);
  raster.setVgap(1);
  raster.setHgap(2);
  hauptPanel = new JPanel(raster);
  hintergrund.add(BorderLayout.CENTER, hauptPanel);

  for (int i = 0; i < 256; i++) {
    JCheckBox c = new JCheckBox();
    c.setSelected(false);
    checkboxListe.add(c);
    hauptPanel.add(c);
  } // Ende der Schleife

  midiEinrichten();

  derFrame.setBounds(50,50,300,300);
  derFrame.pack();
  derFrame.setVisible(true);
} // Methode schließen


public void midiEinrichten() {
  try {
    sequencer = MidiSystem.getSequencer();
    sequencer.open();
    sequence = new Sequence(Sequence.PPQ,4);
    track = sequence.createTrack();
    sequencer.setTempoInBPM(120);

  } catch(Exception e) {e.printStackTrace();}
} // Methode schließen
```

Noch mehr Code für die GUI-Erstellung. Nichts Ungewöhnliches.

Hier werden die Kontrollkästchen erzeugt, auf »false« gesetzt (damit kein Häkchen darin steht) und zur ArrayList UND zum GUI-Panel hinzugefügt.

Der übliche MIDI-Einrichtungskram, der den Sequencer, die Sequence und den Track liefert. Ebenfalls nichts Besonderes.

Hier ist der Mittelpunkt des Geschehens! Wir wandeln den Zustand der Kontrollkästchen in MIDI-Events um und fügen diese dem Track hinzu.

```
public void trackErstellenUndStarten() {
  int[] trackListe = null;

  sequence.deleteTrack(track);
  track = sequence.createTrack();

  for (int i = 0; i < 16; i++) {
    trackListe = new int[16];

    int taste = instrumente[i];

    for (int j = 0; j < 16; j++ ) {

      JCheckBox jc = checkboxListe.get(j + (16*i));
      if (jc.isSelected()) {
        trackListe[j] = taste;
      } else {
        trackListe[j] = 0;
      }
    } // Ende der inneren Schleife

    tracksErzeugen(trackListe);
  } // Ende der äußeren Schleife

  track.add(eventErzeugen(192,9,1,0,16));
  try {

    sequencer.setSequence(sequence);
    sequencer.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);
    sequencer.start();
    sequencer.setTempoInBPM(120);
  } catch(Exception e) {e.printStackTrace();}
} // Methode trackErstellenUndStarten schließen

public class MeinStartListener implements ActionListener {
  public void actionPerformed(ActionEvent a) {
    trackErstellenUndStarten();
  }
} // innere Klasse schließen
```

Wir erzeugen ein Array mit 16 Elementen, das die Werte für ein bestimmtes Instrument über alle 16 Schläge enthält. Wenn das Instrument bei einem bestimmten Schlag spielen soll, ist der Wert des Array-Elements gleich dem >>Tastenschlüssel<< des Instruments. Wenn das Instrument bei diesem Schlag nicht spielen soll, schreiben Sie den Wert 0 hinein.

Der alte Track wird weggeworfen und ein neuer erzeugt.

Dies wird für alle 16 REIHEN ausgeführt (d.h. Bassdrum, Hi-Hat usw.)

Hier wird der >>Tastenschlüssel<< gesetzt, der angibt, um welches Instrument es sich handelt (Bassdrum, Hi-Hat usw.; das Array instrumente enthält die MIDI-Nummern für alle Instrumente).

Dies wird für jeden SCHLAG in der jeweiligen Reihe gemacht.

Ist die Checkbox bei diesem Schlag ausgewählt? Falls ja, setzen Sie den Tastenwert des Instruments in dieses Array-Fach (d.h. in das Fach, das diesen Schlag repräsentiert). Andernfalls soll das Instrument bei diesem Schlag NICHT spielen, also wird eine 0 hineingesetzt.

Hier werden für dieses Instrument und für alle 16 Schläge die Events erzeugt und zum Track hinzugefügt.

Damit wollen wir sicherstellen, dass es am Ende von Schlag 16 auf jeden Fall irgendein Event GIBT. Sonst durchläuft die BeatBox möglicherweise nicht alle 16 Schläge, bevor sie von vorn anfängt. Das hier erzeugte Event hat für unsere BeatBox sonst keinerlei Bedeutung.

Damit können Sie die Anzahl der BeatBox-Schleifeniterationen festlegen bzw. – in diesem Fall – das kontinuierliche Durchlaufen.

JETZT WIRD GESPIELT!!

Die erste der inneren Klassen, die Listener für die Buttons sind. Nichts Besonderes.

```
  public class MeinStoppListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
      sequencer.stop();
    }
  } // innere Klasse schließen

  public class MeinSchnellerListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
      float tempoFactor = sequencer.getTempoFactor();
       sequencer.setTempoFactor((float)(tempoFactor * 1.03));
    }
  } // innere Klasse schließen

  public class MeinLangsamerListener implements ActionListener {
    public void actionPerformed(ActionEvent a) {
      float tempoFactor = sequencer.getTempoFactor();
       sequencer.setTempoFactor((float)(tempoFactor * .97));
    }
  } // innere Klasse schließen
```

Die anderen inneren Klassen – Listener für die Buttons.

Der Tempofaktor passt das Tempo des Sequencers gemäß dem angegebenen Faktor an. Der vorgegebene Wert ist 1.0, die Anpassung entspricht +/– 3% pro Klick.

Hier werden die Events für jeweils ein Instrument erzeugt, und zwar für alle 16 Schläge. Es könnte also beispielsweise ein int[] für die Basstrommel übergeben werden, wobei jeder Index im Array entweder den Tastenschlüssel für dieses Instrument oder eine 0 enthält. Wenn es eine 0 ist, soll das Instrument bei dem entsprechenden Schlag nicht spielen. Andernfalls wird ein Event erzeugt und zum Track hinzugefügt.

```
  public void tracksErzeugen(int[] liste) {

    for (int i = 0; i < 16; i++) {
      int taste = liste[i];

      if (taste != 0) {
        track.add(eventErzeugen(144,9,taste, 100, i));
        track.add(eventErzeugen(128,9,taste, 100, i+1));
      }
    }
  }
```

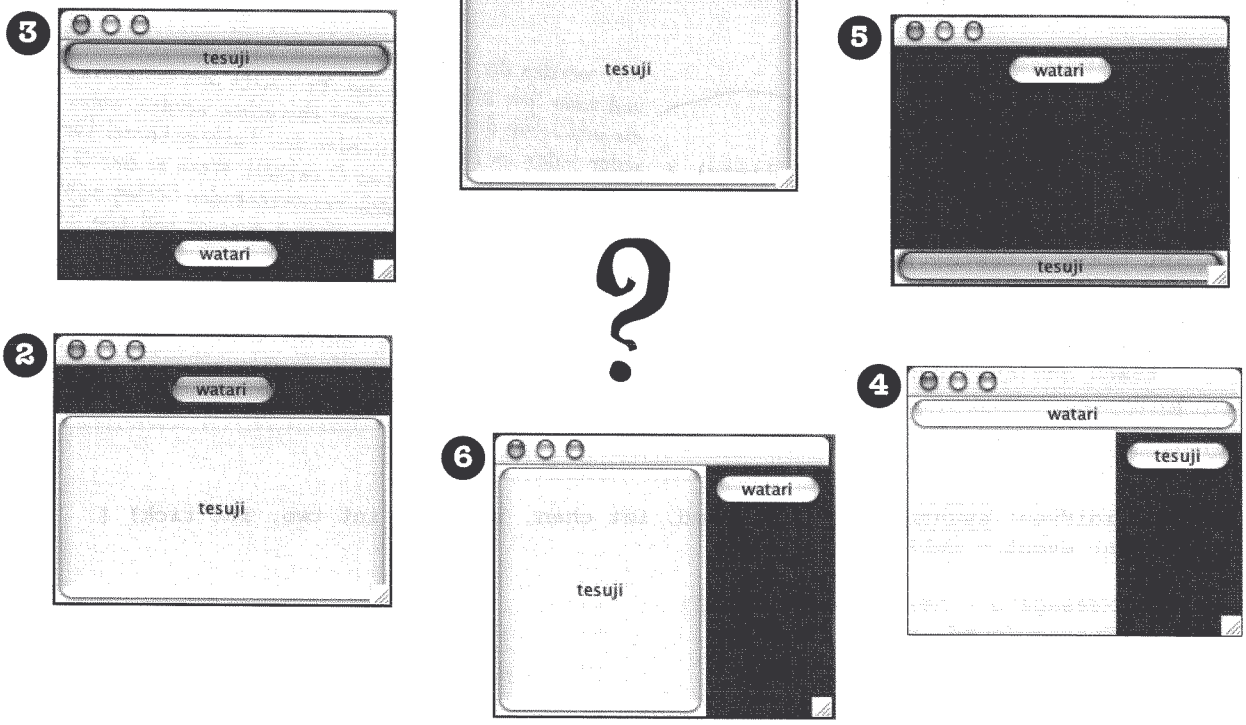
Erzeugt die NOTE ON- und NOTE OFF-Events und fügt sie dem Track hinzu.

```
  public  MidiEvent eventErzeugen(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
      ShortMessage a = new ShortMessage();
      a.setMessage(comd, chan, one, two);
      event = new MidiEvent(a, tick);

    } catch(Exception e) {e.printStackTrace(); }
    return event;
  }

} // Klasse schließen
```

Dies ist die Hilfsmethode aus der CodeKüche des letzten Kapitels. Nichts Neues.

# Welcher Code gehört zu welchem Layout?

Fünf der sechs unten abgebildeten Bildschirmfenster wurden mit einem der Codefragmente auf der gegenüberliegenden Seite erzeugt. Ordnen Sie jedem der fünf Codefragmente das passende Layout zu.



# Code-Fragmente

**D**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton button2 = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH,panel);
panel.add(button2);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

---

**B**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton button2 = new JButton("watari");
panel.add(button2);
frame.getContentPane().add(BorderLayout.CENTER,button);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

---

**C**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton button2 = new JButton("watari");
panel.add(button2);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

---

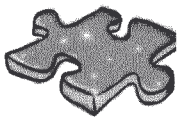
**A**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton button2 = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH,button2);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

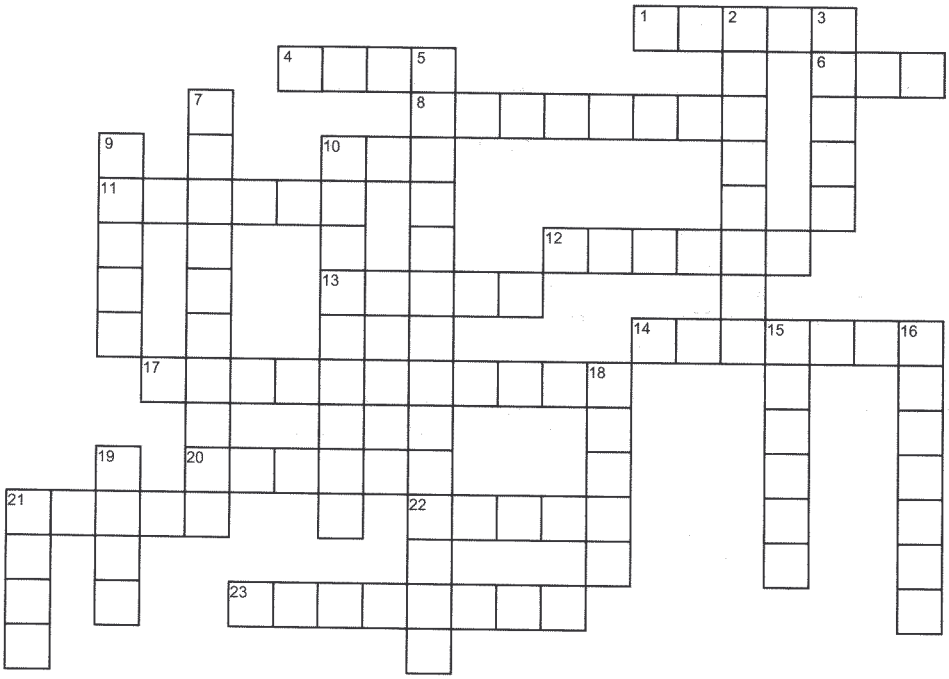
---

**E**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton button2 = new JButton("watari");
frame.getContentPane().add(BorderLayout.SOUTH,panel);
panel.add(button2);
frame.getContentPane().add(BorderLayout.NORTH,button);
```



# GUI-Kreuzworträtsel 7.0



Sie schaffen das.

## Waagerecht

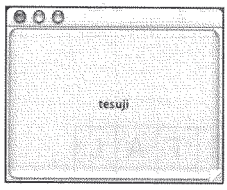
1 Java-Look.
4 MacJava-Look.
6 Ein Widget hinzufügen.
8 Große Auswahl.
10 Vorfahre von Swing.
11 Benutzerverhalten.
12 Border gibt mir den Rest.
13 Wo Hilfe winkt.
14 Kleine Gerätschaften.
17 Regelwerk des Managers.
20 Gefällige Anordnung.
21 Rahmen-Widget.
22 Großes Ereignis.
23 Spitzt seine Ohren.

## Senkrecht

2 Worte, nur Worte ....
3 Kumpel eines Buttons.
5 Wo actionPerformed wohnt.
7 Romulanischer Befehl.
9 Widget im Hintergrund.
10 In Bewegung setzen.
15 Mehr Spaß als Text.
16 Framegröße festlegen.
18 Oberer Borderbereich.
19 Rechtsaußen in Border.
21 Standardlayout im JPanel.

# Lösungen zu den Übungen

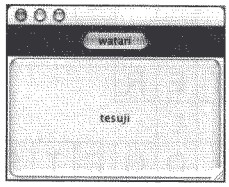
**1**




**C**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton button2 = new JButton("watari");
panel.add(button2);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

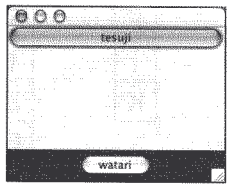
**2**




**D**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton button2 = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH,panel);
panel.add(button2);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

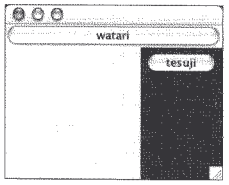
**3**




**E**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton button2 = new JButton("watari");
frame.getContentPane().add(BorderLayout.SOUTH,panel);
panel.add(button2);
frame.getContentPane().add(BorderLayout.NORTH,button);
```

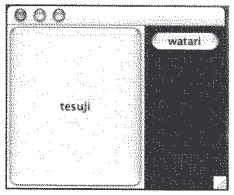
**4**




**A**

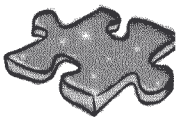
```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton button2 = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH,button2);
frame.getContentPane().add(BorderLayout.EAST,panel);
```

**6**

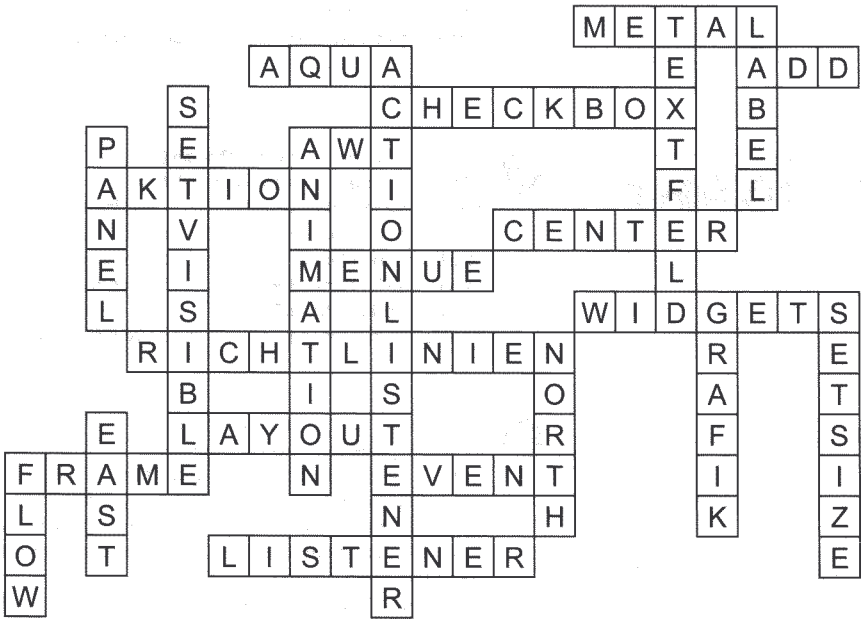



**B**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton button2 = new JButton("watari");
panel.add(button2);
frame.getContentPane().add(BorderLayout.CENTER,button);
frame.getContentPane().add(BorderLayout.EAST,panel);
```



# Lösung zum GUI-Kreuzworträtsel 7.0

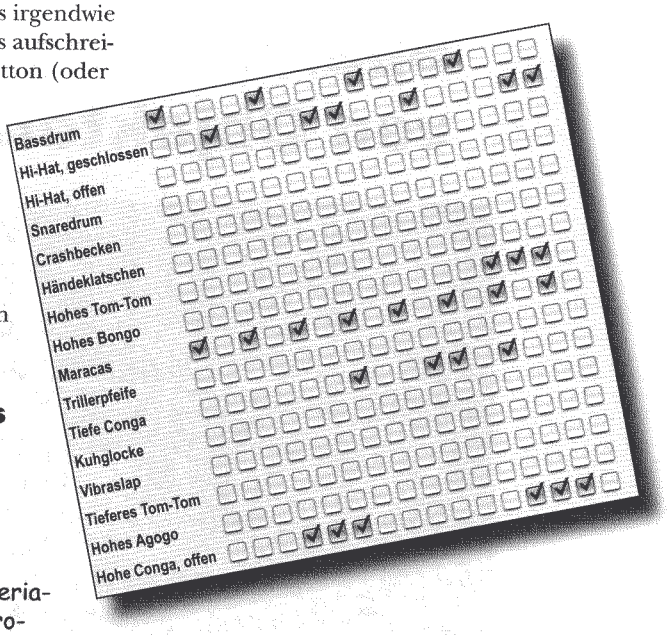


# Speicherung von Objekten

**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*. Was geschieht also, wenn es an der Zeit ist, den Zustand eines Objekts zu *speichern*? Wenn Sie ein Spiel schreiben, brauchen Sie eine Speichern/Wiederherstellen-Funktion. Wenn Sie eine Anwendung schreiben, die Diagramme erzeugt, brauchen Sie eine Speichern/Datei öffnen-Funktion. Muss Ihr Programm Zustände speichern, *können Sie das natürlich auf die mühsame Weise tun* – Sie befragen jedes Objekt und schreiben dann akribisch den Wert jeder Instanzvariablen in eine Datei mit einem selbst definierten Format. Oder **Sie machen es auf die einfache, objektorientierte Weise** – indem Sie lediglich das Objekt selbst gefriertrocknen/flach drücken/speichern/dehydrieren und es mit Wasser anrühren/aufblasen/wiederherstellen/rehydrieren, wenn Sie es zurückhaben wollen. *Manchmal* müssen Sie es allerdings trotzdem noch auf die mühsame Weise machen, vor allem, wenn die von Ihrem Programm gespeicherte Datei durch irgendein Nicht-Java-Programm gelesen werden muss. Wir werden uns deshalb in diesem Kapitel beides ansehen.

# Den Beat einfangen

Sie haben das *perfekte Drum-Pattern* komponiert und möchten es irgendwie festhalten. Sie könnten sich ein Stück Papier nehmen und alles aufschreiben, aber stattdessen drücken Sie einfach auf den ***Speichern***-Button (oder wählen Speichern aus dem Menü Datei). Dann geben Sie ihm einen Namen, wählen ein Verzeichnis aus und atmen auf – mit der Gewissheit, dass sich Ihr Meisterstück nicht beim nächsten Rechnerabsturz in Luft auflöst.




Sie haben eine Reihe von Möglichkeiten, wie Sie den Zustand Ihres Java-Programms speichern können, und für welche davon Sie sich entscheiden, hängt wahrscheinlich davon ab, wie Sie den gespeicherten Zustand *verwenden* wollen. Die folgenden Möglichkeiten werden wir uns in diesem Kapitel ansehen.

## Falls Ihre Daten ausschließlich durch das Java-Programm verwendet werden, mit dem sie erzeugt wurden:

### ① Benutzen Sie Serialisierung

Schreiben Sie eine Datei, die flach gedrückte (serialisierte) Objekte enthält. Dann lassen Sie Ihr Programm die serialisierten Objekte aus der Datei lesen und blasen sie wieder zu lebendigen, atmenden, auf dem Heap wohnenden Objekten auf.

## Falls Ihre Daten von *anderen* Programmen verwendet werden:

### ② Schreiben Sie eine einfache Textdatei

Schreiben Sie eine Datei mit Feldtrennzeichen, die von anderen Programmen ausgewertet werden kann. Zum Beispiel eine Datei mit Tab-Trennzeichen, die von einer Tabellenkalkulation oder einem Datenbankprogramm verwendet werden kann.

Es gibt natürlich noch andere Möglichkeiten. Sie können Daten in jedem gewünschten Format speichern. Statt Zeichen zu schreiben, können Sie Ihre Daten z.B. auch in Byte-Form schreiben. Oder Sie können alle Sorten von primitiven Java-Typen *als* primitive Java-Typen schreiben – es gibt Methoden, um ints, longs, booleans usw. zu schreiben. Abgesehen von der verwendeten Methode sind sich die grundlegenden Techniken für Eingabe/Ausgabe (E/A, engl. I/O für Input/Output) jedoch ziemlich ähnlich: Sie schreiben irgendwelche Daten in *irgendwas* hinein, das in der Regel entweder eine Datei auf einem Speichermedium oder ein Strom aus einer Netzwerkverbindung ist. Beim Lesen der Daten geschieht das Gleiche, nur umgekehrt: Irgendwelche Daten werden entweder aus einer Datei auf einem Speichermedium oder aus einer Netzwerkverbindung gelesen. Und alles, was wir jetzt besprechen, gilt natürlich nur dann, wenn Sie nicht gerade mit einer Datenbank arbeiten.

# Speicherung von Zuständen

Stellen Sie sich vor, Sie haben ein Programm – nehmen wir an, ein Fantasy-Adventure – das man nicht in einer Sitzung zu Ende spielen kann. Im Verlauf des Spiels werden die Spielfiguren stärker, schwächer, klüger usw., und sie beschaffen sich Waffen, die sie benutzen (und verlieren). Sie möchten nicht bei jedem Start des Spiels von Neuem beginnen – schließlich haben Sie ewig gebraucht, um Ihre Figuren für eine spektakuläre Schlacht in Topform zu bringen. Daher brauchen Sie eine Möglichkeit, um den Zustand der Figuren zu speichern bzw. ihn bei der Wiederaufnahme des Spiels wiederherzustellen. Und weil Sie gleichzeitig der Programmierer des Spiels sind, möchten Sie die ganze Speichern- und Wiederherstellen-Geschichte so einfach (und narrensicher) wie möglich haben.

① Möglichkeit eins

**Schreiben Sie die drei serialisierten Spielfigur-Objekte in eine Datei**

Legen Sie eine Datei an und schreiben Sie drei serialisierte Spielfigur-Objekte. Wenn Sie versuchen, diese Datei als Text zu lesen, wird sie für Sie keinen Sinn ergeben:

**¬isrSpielfigur#‰‡,ÖRâIstÃ¤rkeLtyptL java/lang/String;[waffentLjava/lang/String;xp2tElbur[Ljava.lang.String;ÒVçé{GxptBogentSchwerttStaubsq˜ÈtTrolluq˜tbloÃŸe HÃ¤ndetgroÃŸe Axtsq˜xtZauberer uq˜tZauberspr Ã]chetUnsichtbarkeit**

② Möglichkeit zwei

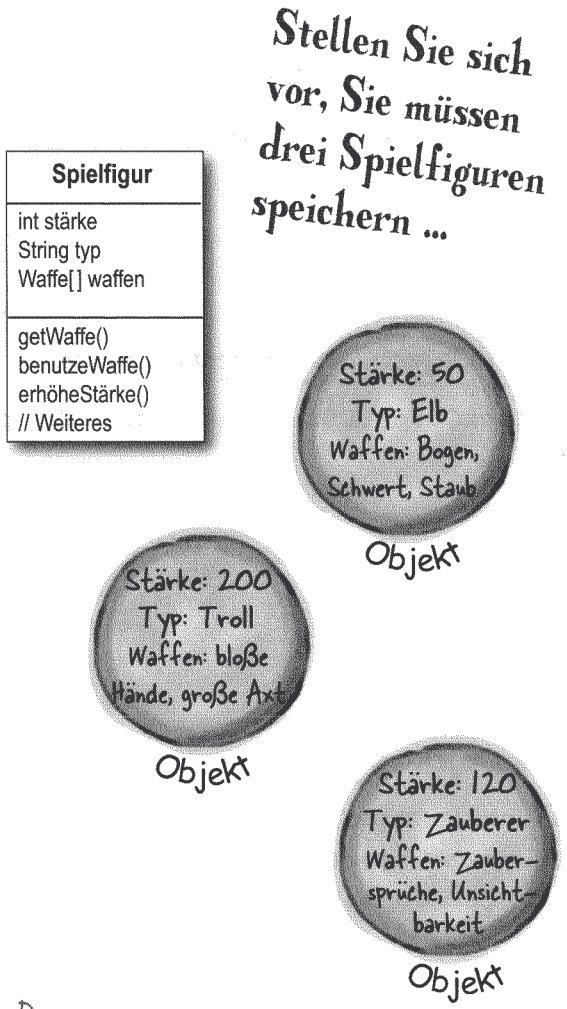
**Schreiben Sie eine einfache Textdatei**

Legen Sie eine Datei an und schreiben Sie drei Zeilen Text, eine pro Spielfigur, mit Kommata zwischen den einzelnen Bestandteilen des Zustands:

**50,Elb,Bogen,Schwert,Staub**

**200,Troll,bloße Hände,große Axt**

**120,Zauberer,Zaubersprüche,Unsichtbarkeit**




Die serialisierte Datei ist zwar für Menschen schwerer zu lesen, aber für Ihr Programm ist es wesentlich einfacher (und sicherer), die drei Objekte aus dem serialisierten Zustand wiederherzustellen statt die in einer Textdatei gespeicherten Werte der Objektvariablen einzulesen. Stellen Sie sich nur vor, wie viele Möglichkeiten es dafür gibt, dass Sie versehentlich die Werte in falscher Reihenfolge einlesen! Aus dem Typ könnte »Staub« geworden sein statt »Elb«, während aus dem Elb eine Waffe wird ...

# Ein serialisiertes Objekt in eine Datei schreiben

Dies sind die Schritte zur Serialisierung (Speicherung) eines Objekts. Machen Sie sich nicht die Mühe, das auswendig zu lernen; wir gehen weiter unten in diesem Kapitel noch genauer darauf ein.

## 1 Einen FileOutputStream erzeugen

```
FileOutputStream fileStream = new FileOutputStream("MeinSpiel.ser");
```

Wenn die Datei >>MeinSpiel.ser<< nicht existiert, wird sie automatisch erzeugt.

Machen Sie ein FileOutputStream-Objekt. FileOutputStream weiß, wie man sich mit einer Datei verbindet (und eine erzeugt).

## 2 Einen ObjectOutputStream erzeugen

```
ObjectOutputStream os = new ObjectOutputStream(fileStream);
```

Mit dem ObjectOutputStream können Sie Objekte schreiben, aber er kann sich nicht direkt mit einer Datei verbinden. Sie müssen ihm einen >>Helfer<< übergeben. Man bezeichnet das auch als >>Verkettung<< eines Stroms mit einem anderen.

## 3 Das Objekt schreiben

```
os.writeObject(figur1);
os.writeObject(figur2);
os.writeObject(figur3);
```

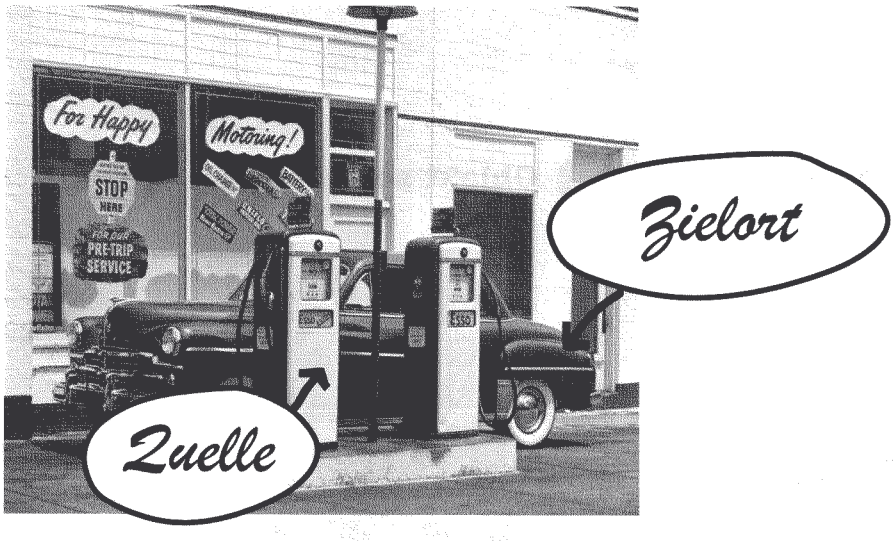
Die von figur1, figur2 und figur3 referenzierten Objekte werden serialisiert und in die Datei >>MeinSpiel.ser<< geschrieben.

## 4 Den ObjectOutputStream schließen

```
os.close();
```

Wenn man den obersten Strom schließt, werden auch die darunter liegenden Ströme geschlossen; der FileOutputStream (und die Datei) schließen sich daher automatisch.

# Daten fließen in Strömen von einem Ort zum anderen.



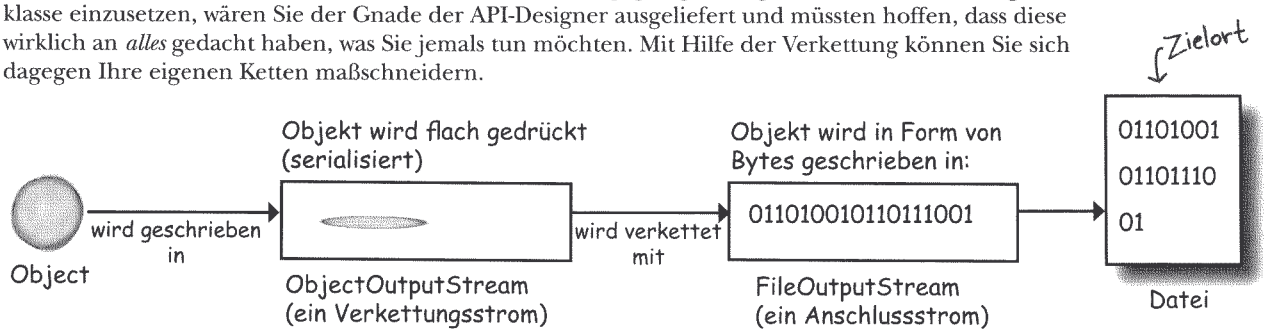

**Anschlussströme repräsentieren eine Verbindung zu einer Quelle oder einem Zielort (einer Datei, einem Socket usw.), während Verkettungsströme selbst keine Verbindung herstellen können und mit einem Anschlussstrom verkettet werden müssen.**

In der Java-E/A-API gibt es ***Anschluss***ströme (connection streams), die Verbindungen zu Zielorten und Quellen wie z.B. Dateien oder Netzwerk-Sockets repräsentieren, und ***Verkettungs***ströme (chain streams), die nur funktionieren, wenn sie mit anderen Strömen verkettet sind.

Oft braucht man mindestens zwei aneinander gehängte Ströme, um damit etwas Nützliches machen zu können – *einen* für den Anschluss und *einen zweiten,* auf dem die Methoden aufgerufen werden. Aber warum zwei? Weil *Anschluss*ströme normalerweise auf einer zu tiefen Stufe stehen (d.h. zu »lowlevel« sind). Ein FileOutputStream (ein Anschlussstrom) beispielsweise besitzt Methoden, um *Bytes* zu schreiben. Aber wir wollen keine *Bytes* schreiben! Wir wollen *Objekte* schreiben, deshalb brauchen wir einen *Verkettungs*strom, der auf einer höheren Stufe steht (also ein »Highlevel«-Strom ist).

Na gut, aber warum gibt es dann nicht einfach *einen* Strom, der *genau das tut,* was Sie wollen? Einen, mit dem Sie Objekte schreiben können, der diese aber unter der Oberfläche in Bytes konvertiert? Denken Sie objektorientiert: Jede Klasse kann genau *eine* Sache gut. FileOutputStreams schreiben Bytes in eine Datei. ObjectOutputStreams wandeln Objekte in Daten um, die in einen Strom geschrieben werden können. Wir erzeugen daher einen FileOutputStream, mit dem wir in eine Datei schreiben können, und an seinem Ende hängen wir einen ObjectOutputStream (einen Verkettungsstrom) an. Wenn wir writeObject() auf dem ObjectOutputStream aufrufen, wird das Objekt in den Strom gepumpt und gelangt von dort in den FileOutputStream, wo es schließlich in Form von Bytes in eine Datei geschrieben wird.

Durch die Möglichkeit, unterschiedliche Kombinationen von Anschluss- und Verkettungsströmen zu bilden, erhalten Sie eine enorme Flexibilität. Wenn Sie dagegen gezwungen wären, nur *eine einzige* Stromklasse einzusetzen, wären Sie der Gnade der API-Designer ausgeliefert und müssten hoffen, dass diese wirklich an *alles* gedacht haben, was Sie jemals tun möchten. Mit Hilfe der Verkettung können Sie sich dagegen Ihre eigenen Ketten maßschneidern.



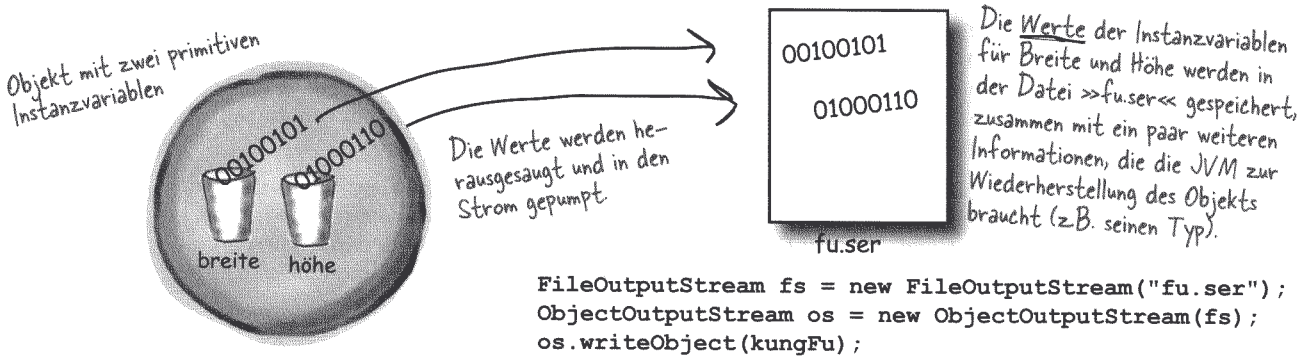
# Was passiert mit einem Objekt bei der Serialisierung wirklich?

## 1 Objekt auf dem Heap

Objekte auf dem Heap haben einen Zustand - den Wert der Instanzvariablen in diesem Objekt. Durch diese Werte unterscheidet sich die Instanz einer Klasse von einer anderen Instanz der gleichen Klasse.

## 2 Objekt serialisiert

Serialisierte Objekte **speichern die Werte der Instanzvariablen**, so dass eine identische Instanz (ein identisches Objekt) in das Leben auf dem Heap zurückgerufen werden kann.




```
Fu kungFu = new Fu();
kungFu.setBreite(37);
kungFu.setHöhe(70);
```

```
FileOutputStream fs = new FileOutputStream("fu.ser");
ObjectOutputStream os = new ObjectOutputStream(fs);
os.writeObject(kungFu);
```

Wir machen einen FileOutputStream, der eine Verbindung zur Datei >>fu.ser<< herstellt, verketten ihn dann mit einem ObjectOutputStream und fordern diesen auf, das Objekt zu schreiben.

# Aber was genau IST der Zustand eines Objekts? Was muss gespeichert werden?

Jetzt wird es langsam interessant. Es ist ja kein Problem, die *primitiven* Werte 37 und 70 zu speichern. Aber was ist, wenn ein Objekt eine Instanzvariable hat, die eine *Referenz* auf ein Objekt ist? Was ist mit einem Objekt mit fünf Instanzvariablen, die Objektreferenzen sind? Und was ist, wenn diese Objektinstanzvariablen selbst Instanzvariablen haben?

Überlegen Sie. Welcher Teil eines Objekts kann einzigartig sein? Was muss wiederhergestellt werden, damit Sie ein Objekt erhalten, das mit dem gespeicherten identisch ist? Natürlich belegt es einen anderen Speicherbereich, aber das macht nichts. Uns interessiert lediglich, dass wir da draußen auf dem Heap ein Objekt mit dem gleichen Zustand bekommen, den das Objekt bei seiner Speicherung hatte.

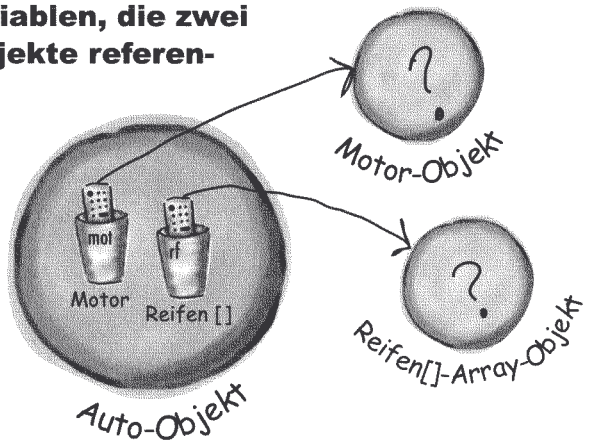
## Gewichtheben fürs Gehirn

Wie kann das Auto-Objekt so gespeichert werden, dass sein Originalzustand wiederhergestellt werden kann?

Überlegen Sie, was Sie wohl brauchen, um das Auto zu speichern – und wie Sie vorgehen würden.

Und was passiert, wenn ein Motor-Objekt eine Referenz auf einen Vergaser hat? Was steckt in dem Reifen[]-Array-Objekt?

**Das Auto-Objekt hat zwei Instanzvariablen, die zwei andere Objekte referenzieren.**



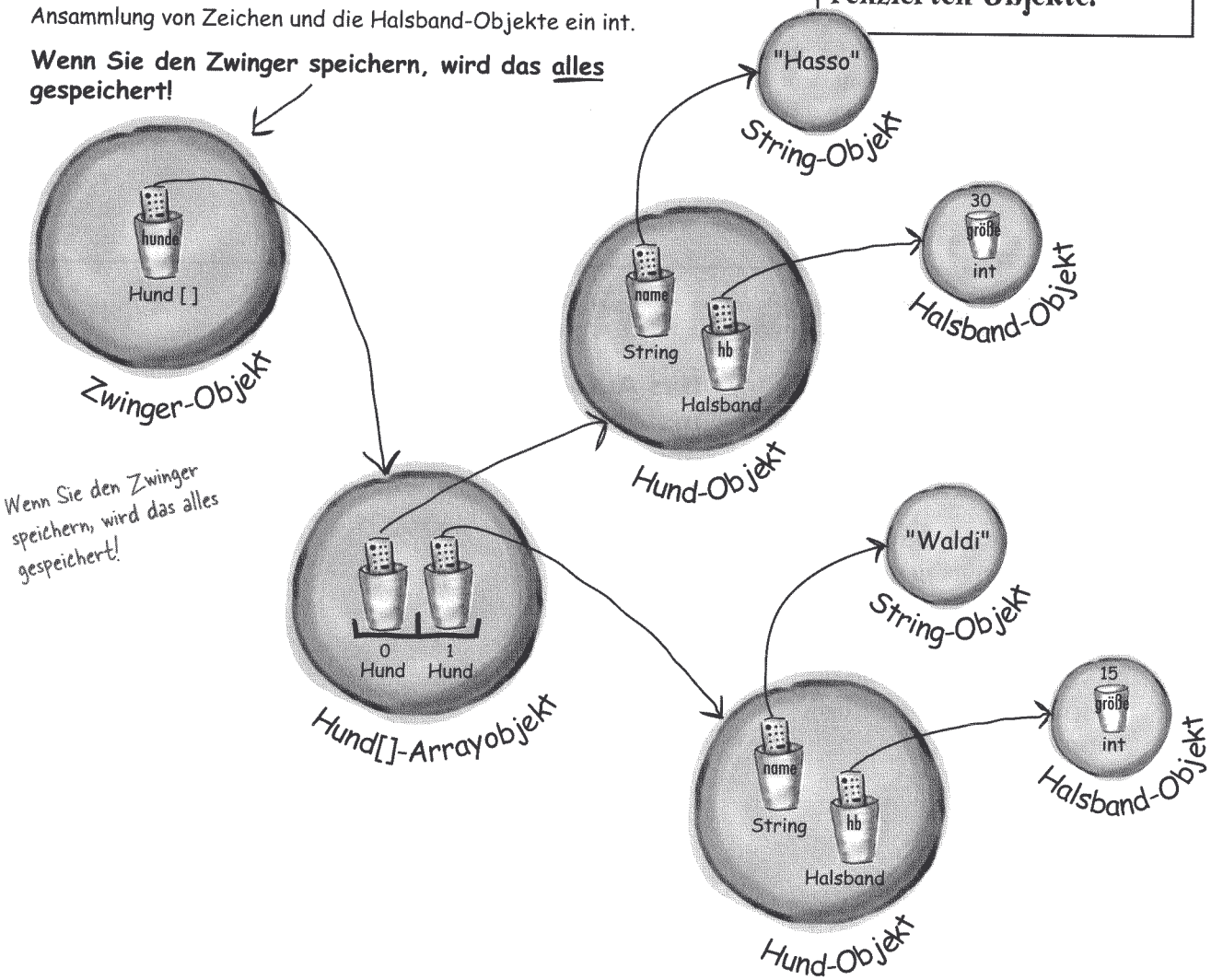

**Was ist nötig, um ein Auto-Objekt zu speichern?**

**Wenn ein Objekt serialisiert wird, werden alle Objekte, die es über Instanzvariablen referenziert, *ebenfalls* mit serialisiert. Und all die Objekte, die *diese* Objekte referenzieren, werden serialisiert. Und all die Objekte, die *diese* Objekte referenzieren, werden serialisiert ... und das Beste daran ist, das alles passiert automatisch!**

Bei der Serialisierung wird der vollständige Objektgraph gespeichert. Zuerst das zu serialisierende Objekt und ausgehend davon alle über Instanzvariablen referenzierten Objekte.

Dieses Zwinger-Objekt hat eine Referenz auf ein Hund[]-Array-Objekt. Hund[] wiederum enthält Referenzen auf zwei Hund-Objekte. Jedes Hund-Objekt enthält eine Referenz auf einen String und ein Halsband-Objekt. Die String-Objekte enthalten eine Ansammlung von Zeichen und die Halsband-Objekte ein int.

**Wenn Sie den Zwinger speichern, wird das alles gespeichert!**



# Wenn Sie wollen, dass Ihre Klasse serialisierbar ist, implementieren Sie Serializable

Das Interface Serializable wird auch als *Marker-* oder *Tag*-Interface bezeichnet, da es keinerlei zu implementierende Methoden enthält. Es dient nur dem einen Zweck, die Klasse, von der es implementiert wird, als *serialisierbar* zu markieren. Mit anderen Worten: Objekte dieses Typs können über den Serialisierungsmechanismus gespeichert werden. Wenn irgendeine Superklasse einer Klasse serialisierbar ist, ist auch die Unterklasse automatisch serialisierbar, selbst wenn diese nicht ausdrücklich *implements Serializable* deklariert. (So funktionieren Interfaces eben. Wenn für Ihre Superklasse gilt, sie IST-EIN Serializable, dann sind Sie es auch.)

```
objectOutputStream.writeObject(meineSchachtel);
```

Was auch immer hier steht, es MUSS Serializable implementieren, sonst wird das zur Laufzeit scheitern.

---

```
import java.io.*;

public class Schachtel implements Serializable {

  private int breite;
  private int höhe;

  public void setBreite(int b) {
    breite = b;
  }

  public void setHöhe(int h) {
    höhe = h;
  }

  public static void main(String[] args) {

    Schachtel meineSchachtel = new Schachtel();
    meineSchachtel.setBreite(50);
    meineSchachtel.setHöhe(20);

    try {
      FileOutputStream fs = new FileOutputStream("foo.ser");
      ObjectOutputStream os = new ObjectOutputStream(fs);
      os.writeObject(meineSchachtel);
      os.close();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Serializable gehört zum E/A-Paket java.io, daher brauchen Sie den Import.

Keine Methoden zu implementieren, aber wenn Sie >>implements Serializable<< schreiben, sagen Sie damit der JVM: >>Es ist okay, wenn Objekte dieses Typs serialisiert werden.<<

Diese beiden Werte werden gespeichert.

E/A-Operationen können Exceptions auslösen.

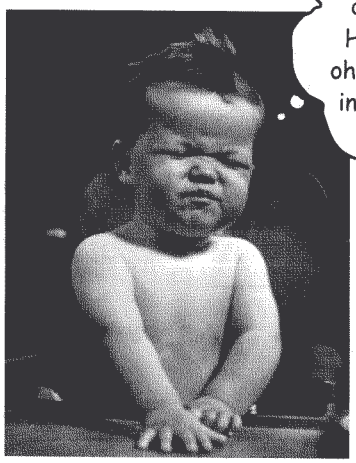
Stellen Sie eine Verbindung zu einer Datei namens >>foo.ser<< her, falls diese existiert. Falls nicht, legen Sie eine neue Datei namens >>foo.ser<< an.

Erzeugen Sie einen ObjectOutputStream, der mit dem Anschlussstrom verkettet ist.

Lassen Sie ihn das Objekt schreiben.

## Serialisierung heißt: alles oder nichts.

**Können Sie sich vorstellen, was passieren würde, wenn ein Teil des Objektzustands nicht korrekt gespeichert würde?**



Huuu! Es gruselt mich schon, wenn ich nur daran denke! Wenn zum Beispiel ein Hund ohne Gewicht zurückkäme. Oder ohne Ohren. Oder das Halsband kommt in Größe 3 zurück statt in Größe 30. Das darf einfach nicht zulässig sein!

**Entweder wird der gesamte Objektgraph korrekt serialisiert, oder die Serialisierung schlägt fehl.**

**Sie können ein Teich-Objekt nicht serialisieren, wenn sich seine Ente-Instanzvariable weigert, serialisiert zu werden (indem sie Serializable nicht implementiert).**

```
import java.io.*;

public class Teich implements Serializable {

  private Ente ente = new Ente();

  public static void main(String[] args) {
    Teich meinTeich = new Teich();
    try {
      FileOutputStream fs = new FileOutputStream("teich.ser");
      ObjectOutputStream os = new ObjectOutputStream(fs);

      os.writeObject(meinTeich);
      os.close();

    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Teichobjekte können serialisiert werden.

Die Klasse Teich hat nur eine einzige Instanzvariable, eine Ente.

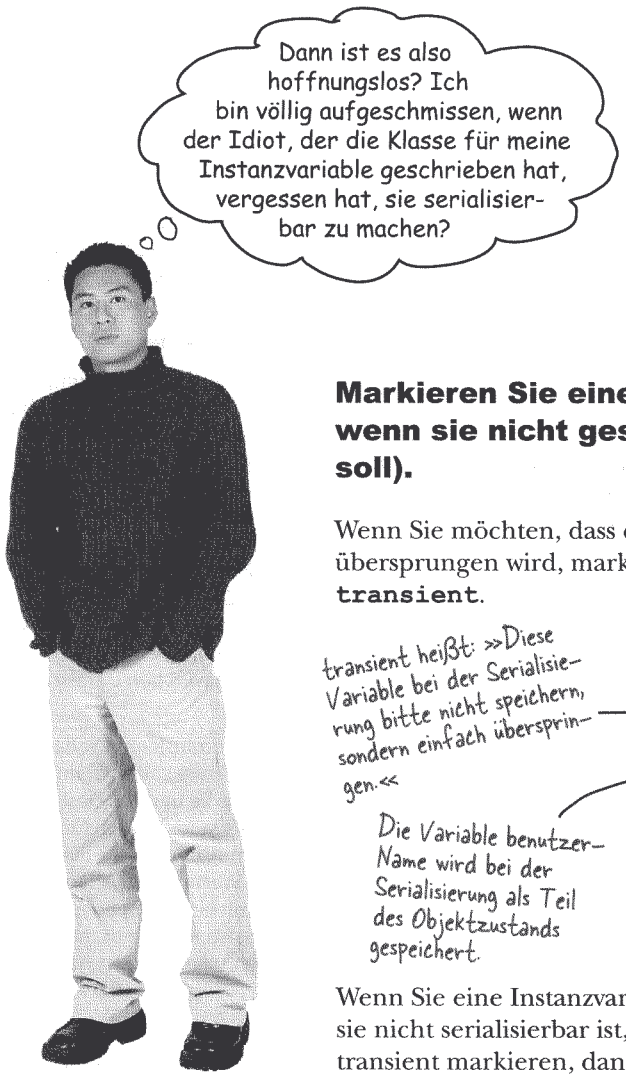
Wenn Sie meinTeich (ein Teich-Objekt) serialisieren, wird seine Ente-Instanzvariable automatisch mit serialisiert.

```
public class Ente {
  // hier steht der Entencode
}
```

Oh Schreck! Ente ist nicht serialisierbar! Sie implementiert Serializable nicht, d.h., der Versuch, das Teichobjekt zu serialisieren, scheitert, weil die Ente-Instanzvariable des Teichs nicht gespeichert werden kann.

Wenn Sie versuchen, main in der Klasse Teich auszuführen:

```
Datei Bearbeiten Fenster Hilfe IchBedaure
% java Teich
java.io.NotSerializableException: Ente
        at Teich.main(Teich.java:13)
```




## Markieren Sie eine Instanzvariable als transient, wenn sie nicht gespeichert werden kann (oder soll).

Wenn Sie möchten, dass eine Instanzvariable beim Serialisierungsvorgang übersprungen wird, markieren Sie die Variable mit dem Schlüsselwort **transient**.

transient heißt: »Diese Variable bei der Serialisierung bitte nicht speichern, sondern einfach überspringen.«

Die Variable benutzerName wird bei der Serialisierung als Teil des Objektzustands gespeichert.

```
import java.net.*;
class Chat implements Serializable {
  transient String aktuelleID;

  String benutzerName

  // weiterer Code
}
```

Wenn Sie eine Instanzvariable haben, die nicht gespeichert werden kann, weil sie nicht serialisierbar ist, können Sie diese Variable mit dem Schlüsselwort transient markieren, dann wird sie beim Serialisierungsprozess einfach übersprungen.

Aber aus welchem Grund sollte eine Variable nicht serialisierbar sein? Denkbar ist z.B., dass der Programmierer der Klasse schlicht und einfach *vergessen* hat, die Klasse Serializable implementieren zu lassen. Vielleicht hängt das Objekt aber auch von laufzeitspezifischen Informationen ab, die man einfach nicht speichern kann. Zwar sind die meisten Dinge in den Java-Klassenbibliotheken serialisierbar, doch kann z.B. so etwas wie eine Netzwerkverbindung, ein Thread oder ein Dateiobjekt nicht gespeichert werden. Diese hängen von bestimmten zur Laufzeit gesammelten »Erfahrungen« ab (sind also laufzeitspezifisch). Mit anderen Worten, sie werden auf eine Art und Weise instantiiert, die spezifisch dafür ist, wie Ihr Programm jetzt gerade ausgeführt wird, d.h. auf einer ganz bestimmten Plattform und mit einer ganz bestimmten JVM. Wenn das Programm geschlossen wurde, kann man diese Dinge nicht in sinnvoller Weise wieder zum Leben erwecken – sie müssen jedes Mal von Grund auf neu erzeugt werden.

## Es gibt keine Dummen Fragen

**F: Wenn Serialisierung so wichtig ist, warum ist das nicht standardmäßig für alle Klassen vorgegeben? Warum wird Serializable nicht von der Klasse Object implementiert, so dass alle Unterklassen automatisch serialisierbar werden?**

A: Auch wenn die meisten Klassen Serializable implementieren werden – und sollten –, haben Sie doch immer die Wahl. Und Sie müssen Klasse für Klasse bewusst entscheiden (für jede Klasse, die Sie schreiben), ob die Serialisierung durch die Implementierung von Serializable »eingeschaltet« werden soll. Zunächst einmal: Wenn die Serialisierbarkeit vorgegeben wäre – wie würden Sie sie abschalten? Interfaces sind ein Merkmal für Funktionalität, nicht für *fehlende* Funktionalität. Das Modell der Polymorphie würde daher nicht korrekt funktionieren, wenn Sie sagen müssten: »implements NonSerializable«, um zu verkünden, dass Sie nicht gespeichert werden können.

**F: Warum sollte ich jemals eine *nicht* serialisierbare Klasse schreiben?**

A: Es gibt nur ganz wenige Gründe dafür, aber Sie könnten beispielsweise ein Sicherheitsproblem haben, bei dem Sie die Speicherung eines Kennwort-Objekts verhindern wollen. Oder Sie haben vielleicht ein Objekt, dessen Speicherung keinen Sinn ergibt, weil seine wichtigsten Instanzvariablen selbst nicht serialisierbar sind, somit können Sie *Ihre* Klasse nicht auf sinnvolle Weise serialisierbar machen.

**F: Wenn eine Klasse, die ich verwende, nicht serialisierbar ist, es aber dafür keinen guten Grund gibt (außer dass der Programmierer es einfach vergessen hat oder zu blöd war) –, kann ich dann eine Unterklasse dieser »bösen« Klasse schreiben und diese *Unterklasse* serialisierbar machen?**

A: Ja! Wenn die Klasse selbst erweitert werden kann (d.h. nicht final ist), können Sie eine serialisierbare Unterklasse schreiben und diese einfach überall dort einsetzen, wo Ihr Code den Typ der Superklasse erwartet. (Wie Sie wissen, wird das durch Polymorphie gestattet.) Was ein weiteres interessantes Problem aufwirft: Was *bedeutet* es, wenn die Superklasse nicht serialisierbar ist?

**F: Und? Was *bedeutet* es denn nun, wenn man eine serialisierbare Unterklasse einer nicht serialisierbaren Superklasse hat?**

A: Dafür müssen wir uns erst einmal ansehen, was bei der Deserialisierung einer Klasse passiert (das besprechen wir auf den folgenden Seiten). Kurz gesagt: Wenn ein Objekt deserialisiert wird und seine Superklasse *nicht* serialisierbar ist, wird der Konstruktor der Superklasse ausgeführt, als würde ein neues Objekt dieses Typs erzeugt. Wenn es keinen guten Grund für eine Klasse gibt, nicht serialisierbar zu sein, kann die Erstellung einer serialisierbaren Unterklasse eine gute Lösung sein.

**F: Halt! Gerade ist mir was Wichtiges aufgefallen ... wenn man eine Variable »transient« macht, heißt das doch, dass die Variable bei der Serialisierung übersprungen wird. Was passiert dann damit? Wenn wir eine nicht serialisierbare Instanzvariable haben, lösen wir das Problem, indem wir die Instanzvariable transient machen – aber brauchen wir diese Variable denn nicht, wenn das Objekt wieder zum Leben erweckt wird? Mit anderen Worten, geht es bei der Serialisierung nicht vor allem darum, den Zustand eines Objekts zu konservieren?**

A: Ja, das ist ein Problem, aber glücklicherweise gibt es eine Lösung. Wenn Sie ein Objekt serialisiert haben, bekommen Sie eine transiente Referenz-Instanzvariable als *null* zurück, ganz egal, welchen Wert sie zum Zeitpunkt der Speicherung hatte. Das bedeutet, dass der gesamte Objektgraph, der an diese spezielle Instanzvariable gebunden ist, nicht gespeichert wird. Das könnte natürlich böse Folgen haben, denn Sie brauchen ja wahrscheinlich für diese Variable einen anderen Wert als null.

Sie haben zwei Möglichkeiten:

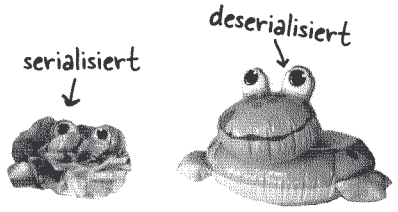
1) Wenn Sie das Objekt zurückbekommen, reinitialisieren Sie diese null-Instanzvariable wieder mit irgendeinem Standardzustand. Das funktioniert, wenn Ihr deserialisiertes Objekt nicht von einem bestimmten Wert für die transiente Variable abhängt. Mit anderen Worten: Es könnte wichtig sein, dass der Hund ein Halsband hat, aber vielleicht sind alle Halsband-Objekte gleich, so dass es egal ist, wenn Sie dem ins Leben zurückgerufenen Hund ein nagelneues Halsband verpassen – niemand wird den Unterschied bemerken.

2) Wenn der Wert der transienten Variablen aber doch eine Rolle spielt (wenn beispielsweise die Farbe und das Design des transienten Halsbands typisch für den jeweiligen Hund sind), müssen Sie die charakteristischen Werte des Halsbands speichern und damit ein nagelneues, aber mit dem Originalhalsband identisches Halsband erzeugen, wenn Sie den Hund wiederbeleben.

**F: Was passiert, wenn es sich bei zwei Objekten im Objektgraphen um dasselbe Objekt handelt? Wenn beispielsweise zwei verschiedene Katzen-Objekte im Zwinger sind, aber beide Katzen eine Referenz auf dasselbe Besitzer-Objekt haben? Wird der Besitzer zweimal gespeichert? Ich hoffe, nicht.**

A: Eine ausgezeichnete Frage! Die Serialisierung ist schlau genug, es zu wissen, wenn zwei Objekte im Graphen ein und dasselbe Objekt sind. In diesem Fall wird das Objekt nur ein Mal gespeichert und bei der Deserialisierung werden alle Referenzen auf dieses eine Objekt wiederhergestellt.

# Deserialisierung: Wiederherstellung eines Objekts




Das Wichtigste an der Serialisierung eines Objekts ist, dass Sie es später wieder in seinen ursprünglichen Zustand zurücksetzen können während eines anderen Laufs der virtuellen Maschine (das muss noch nicht einmal die gleiche JVM sein, die zum Zeitpunkt der Serialisierung des Objekts lief). In vieler Hinsicht ist Deserialisierung wie umgekehrte Serialisierung.

## 1 Einen FileInputStream erzeugen

Wenn die Datei >>MeinSpiel.ser<< nicht existiert, wird sie automatisch erzeugt.

```
FileInputStream fileStream = new FileInputStream("MeinSpiel.ser");
```

Erzeugen Sie ein FileInputStream-Objekt. FileInputStream weiß, wie man sich mit einer vorhandenen Datei verbindet.

## 2 Einen ObjectInputStream erzeugen

```
ObjectInputStream os = new ObjectInputStream(fileStream);
```

Mit dem ObjectInputStream können Sie Objekte lesen, aber keine direkte Verbindung zu einer Datei herstellen. Sie müssen ihn mit einem Anschlussstrom verketten, in diesem Fall mit einem FileInputStream.

## 3 Die Objekte lesen

```
Object eins = os.readObject();
Object zwei = os.readObject();
Object drei = os.readObject();
```

Jedes Mal, wenn Sie readObject() aufrufen, erhalten Sie das nächste Objekt aus dem Strom. Sie lesen sie also in der gleichen Reihenfolge wieder ein, in der sie geschrieben wurden. Wenn Sie versuchen, mehr Objekte zu lesen, als Sie geschrieben haben, erhalten Sie eine dicke fette Exception.

## 4 Die Objekte casten

```
Spielfigur elb = (Spielfigur) eins;
Spielfigur troll = (Spielfigur) zwei;
Spielfigur zauberer = (Spielfigur) drei;
```

Der Rückgabewert von readObject() ist vom Typ Object, daher müssen Sie dieses durch einen Cast wieder in den Typ umwandeln, der es Ihrer Kenntnis nach ist.

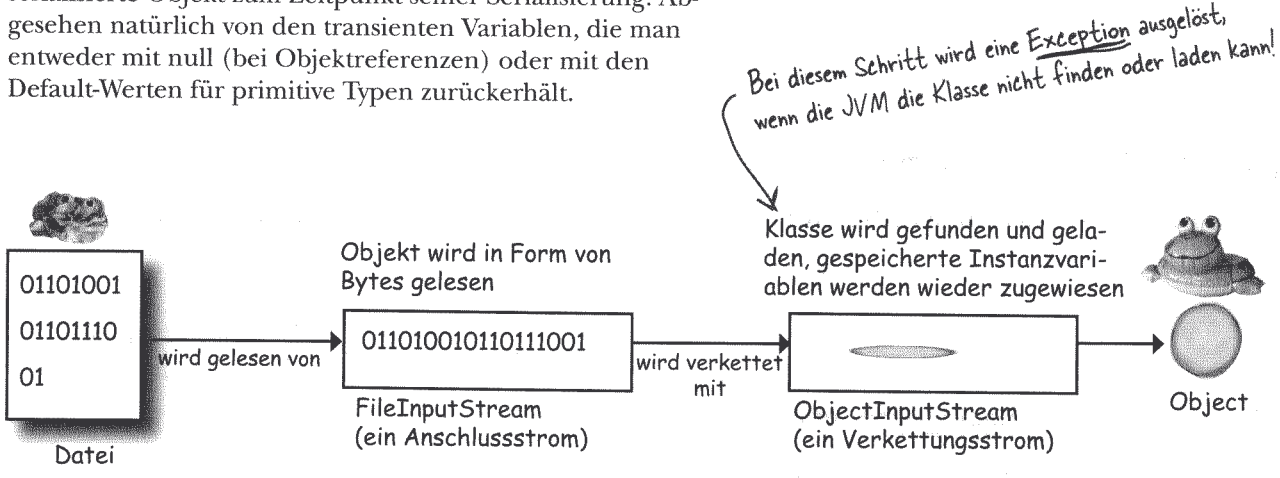
## 5 Den ObjectInputStream schließen

```
os.close();
```

Wenn man den oberen Strom schließt, werden auch die darunter liegenden geschlossen; der FileInputStream (und die Datei) schließen sich daher automatisch.

# Was passiert bei der Deserialisierung?

Bei der Deserialisierung versucht die JVM das Objekt ins Leben zurückzuholen, indem sie ein neues Objekt auf dem Heap erzeugt, das den gleichen Zustand hat wie das serialisierte Objekt zum Zeitpunkt seiner Serialisierung. Abgesehen natürlich von den transienten Variablen, die man entweder mit null (bei Objektreferenzen) oder mit den Default-Werten für primitive Typen zurückerhält.




1. Das Objekt wird aus dem Strom **gelesen**.

2. Die JVM bestimmt die **Klasse** des Objekts (anhand der mit dem serialisierten Objekt gespeicherten Informationen).

3. Die JVM versucht, die **Klasse** des Objekts **zu finden und zu laden**. Wenn die JVM die Klasse nicht finden und/oder laden kann, löst sie eine Exception aus, und die Deserialisierung schlägt fehl.

4. Einem neuen Objekt wird Platz auf dem Heap zugeteilt, aber der **Konstruktor des serialisierten Objekts wird NICHT ausgeführt**! Bei seiner Ausführung würde ja der ursprüngliche »new«-Zustand des Objekts wiederhergestellt, und das ist nicht das, was wir wollen. Wir wollen das Objekt wieder in dem Zustand haben, den es *bei der Serialisierung* hatte, nicht bei seiner erstmaligen Erzeugung.

5. Wenn irgendwo weiter oben im Vererbungsbaum des Objekts eine nicht serialisierbare Klasse vorkommt, wird der **Konstruktor für diese nicht serialisierbare Klasse ausgeführt** und zudem alle darüber stehenden Konstruktoren (selbst wenn sie serialisierbar sind). Wenn die Konstruktorkette einmal in Gang gesetzt ist, kann man sie nicht mehr aufhalten. Das bedeutet, dass alle Superklassen - angefangen bei der ersten nicht serialisierbaren - ihren Zustand reinitialisieren.

6. **Die Instanzvariablen des Objekts erhalten die Werte aus dem serialisierten Zustand.** Transiente Variablen erhalten den Wert null bei Objektreferenzen bzw. den jeweiligen Default-Wert für primitive Typen (0, false usw.).

## Es gibt keine Dummen Fragen

**F: Warum wird die Klasse nicht als Teil des Objekts mit gespeichert? Dann hätte man doch nicht das Problem, ob die Klasse gefunden werden kann.**

A: Sicher, so hätte man Serialisierung auch einrichten können. Aber was für eine enorme Verschwendung wäre das! Vielleicht wäre das noch nicht einmal ein Problem, solange Sie Serialisierung nur dazu benutzen, Objekte in eine Datei oder auf eine lokale Festplatte zu schreiben. Aber Serialisierung wird auch dazu benutzt, Objekte über eine Netzwerkverbindung zu versenden. Würde zu jedem serialisierten (versendbaren) Objekt auch noch eine Klasse gepackt, wäre die Bandbreite ein noch viel größeres Problem, als sie schon ist.

Wenn man serialisierte Objekte über ein Netzwerk versenden will, gibt es jedoch einen Mechanismus, bei dem man das serialisierte Objekt mit einer URL »stempeln« kann; diese gibt darüber Auskunft, wo die Klasse zu finden ist. Zum Einsatz kommt dies bei der Java Remote Method Invocation (RMI, Fernaufruf von Methoden); dabei können Sie beispielsweise ein serialisiertes Objekt als Methodenargument versenden. Wenn dann die JVM, die den Aufruf empfängt, die Klasse nicht selbst hat, kann sie diese mit Hilfe der URL automatisch aus dem Netz holen und laden. (Wir besprechen RMI in Kapitel 18.)

**F: Was ist mit statischen Variablen? Werden die serialisiert?**

A: Nein. Erinnern Sie sich, statisch bedeutet »einer pro Klasse«, nicht »einer pro Objekt«. Statische Variablen werden nicht gespeichert, und wenn ein Objekt deserialisiert wird, hat es genau die statischen Variablen, die seine Klasse *zu dem Zeitpunkt* hat. Die Moral: Machen Sie serialisierbare Objekte nicht von einer sich dynamisch ändernden statischen Variablen abhängig! Sie könnte nicht mehr die gleiche sein, wenn Ihr Objekt zurückkommt.

# Speicherung und Wiederherstellung der Spielfiguren

```
import java.io.*;

public class SpielSpeicherungTest {
  public static void main(String[] args) {
    Spielfigur eins = new Spielfigur(50, "Elb", new String[] { "Bogen", "Schwert", "Staub" });
    Spielfigur zwei = new Spielfigur(200, "Troll", new String[] {"bloße Hände", "große Axt" });
    Spielfigur drei = new Spielfigur(120, "Zauberer", new String[] {"Zaubersprüche", "Unsichtbarkeit" });

    // der Code, der hier steht, soll mit den Figuren etwas machen, was ihre Zustandswerte ändern könnte

    try {
      ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream("Spiel.ser"));
      os.writeObject(eins);
      os.writeObject(zwei);
      os.writeObject(drei);
      os.close();
    } catch (IOException ex) {
      ex.printStackTrace();
    }
    eins = null;
    zwei = null;
    drei = null;

    try {
      ObjectInputStream is = new ObjectInputStream(new FileInputStream("Spiel.ser"));
      Spielfigur einsWiederhergestellt = (Spielfigur) is.readObject();
      Spielfigur zweiWiederhergestellt = (Spielfigur) is.readObject();
      Spielfigur dreiWiederhergestellt = (Spielfigur) is.readObject();
      isClose();

      System.out.println("Typ 1. Figur: " + einsWiederhergestellt.getTyp());
      System.out.println("Typ 2. Figur: " + zweiWiederhergestellt.getTyp());
      System.out.println("Typ 3. Figur: " + dreiWiederhergestellt.getTyp());
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

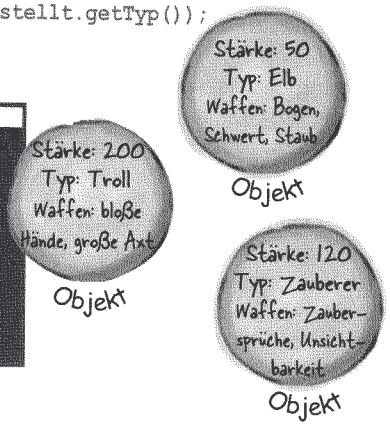
Machen Sie ein paar Figuren ...

Wir setzen sie auf null, so dass wir auf die Objekte auf dem Heap nicht mehr zugreifen können.

Jetzt lesen wir sie aus der Datei wieder ein ...

Hier die Überprüfung, ob es funktioniert hat.

```
Datei Bearbeiten Fenster Hilfe Wiederbelebung
% java SpielSpeicherungTest
Typ 1. Figur: Elb
Typ 2. Figur: Troll
Typ 3. Figur: Zauberer
```



## Die SpielFigur-Klasse

```
import java.io.*;

public class Spielfigur implements Serializable {
  int stärke;
  String typ;
  String[] waffen;

  public Spielfigur(int s, String t, String[] w) {
    stärke = s;
    typ = t;
    waffen = w;
  }

  public int getStärke() {
    return stärke;
  }

  public String getTyp() {
    return typ;
  }

  public String getWaffen() {
    String waffenListe = "";

    for (int i = 0; i < waffen.length; i++) {
      waffenListe += waffen[i] + " ";
    }
    return waffenListe;
  }
}
```

Diese Basisklasse dient nur zum Testen der Serialisierung; und wir haben hier auch kein richtiges Spiel, sondern überlassen das Ihrer Experimentierfreudigkeit.

# Serialisierung von Objekten

## Punkt für Punkt

- Sie können den Zustand eines Objekts speichern, indem Sie es serialisieren.
- Um ein Objekt zu serialisieren, brauchen Sie einen ObjectOutputStream (aus dem Paket java.io).
- Ströme sind entweder Anschlussströme oder Verkettungsströme.
- Anschlussströme können eine Verbindung mit einer Quelle oder mit einem Zielort repräsentieren, typischerweise mit einer Datei, einer Netzwerk-Socket-Verbindung oder der Konsole.
- Verkettungsströme können keine Verbindung mit einer Quelle oder einem Zielort herstellen und müssen mit einem Anschlussstrom (oder einem anderen Strom) verkettet werden.
- Um ein Objekt als Datei zu serialisieren, erzeugen Sie einen FileOutputStream und verketten ihn mit einem ObjectOutputStream.
- Um ein Objekt zu serialisieren, rufen Sie *writeObject(dasObjekt)* auf dem ObjectOutputStream auf. Auf dem FileOutputStream brauchen Sie keine Methoden aufzurufen.
- Um serialisiert werden zu können, muss ein Objekt das Interface Serializable implementieren. Wenn eine Superklasse dieser Klasse Serializable implementiert, ist die Unterklasse automatisch serialisierbar, selbst wenn sie nicht ausdrücklich *implements Serializable* deklariert.
- Wird ein Objekt serialisiert, wird sein gesamter Objektgraph serialisiert. Das heißt, dass auch alle Objekte, die durch die Instanzvariablen des serialisierten Objekts referenziert werden, serialisiert werden, ebenso alle Objekte, die von diesen Objekten referenziert werden ... usw.
- Ist irgendein Objekt in dem Graphen nicht serialisierbar, wird zur Laufzeit eine Exception ausgelöst, es sei denn, die Instanzvariable, die das Objekt referenziert, wird übersprungen.
- Wenn Sie möchten, dass eine Instanzvariable bei der Serialisierung übersprungen wird, markieren Sie sie mit dem Schlüsselwort *transient*. Die Variable wird als null (bei Objektreferenzen) bzw. mit dem vordefinierten Wert (bei primitiven Typen) wiederhergestellt.
- Während der Deserialisierung müssen der JVM die Klassen aller Objekte im Graphen zur Verfügung stehen.
- Objekte werden in der Reihenfolge gelesen (mit readObject()), in der sie ursprünglich geschrieben wurden.
- Der Rückgabetyp von readObject() ist Object, daher brauchen deserialisierte Objekte einen Cast auf ihren wirklichen Typ.
- Statische Variablen werden nicht serialisiert! Es ergibt keinen Sinn, den Wert einer statischen Variablen als Teil des Zustands von einem bestimmten Objekt zu speichern, da alle Objekte dieses Typs einen einzigen Wert gemeinsam benutzen – den in der Klasse.

# Einen String in eine Textdatei schreiben

Das Speichern von Objekten durch Serialisierung ist der einfachste Weg, Daten zwischen den Durchläufen eines Java-Programms zu speichern und wiederherzustellen. Manchmal müssen Sie jedoch auch Daten in einer guten alten Textdatei speichern. Stellen Sie sich vor, Ihr Java-Programm muss Daten in eine einfache Textdatei schreiben, die ein anderes Programm (vielleicht ein Nicht-Java-Programm) lesen muss. So könnte beispielsweise ein Servlet (Java-Code, der auf Ihrem Webserver läuft) Formulardaten empfangen, die der Benutzer in einem Browser eingegeben hat, und diese Daten könnte es in eine Textdatei schreiben, die jemand anderer in eine Tabellenkalkulaton lädt, um sie zu analysieren.

Das Schreiben von Textdaten (eigentlich: das Schreiben eines Strings) ähnelt dem Schreiben eines Objekts, abgesehen davon, dass Sie eben einen String schreiben statt eines Objekts und einen FileWriter verwenden statt eines FileOutputStreams (und er wird auch nicht mit einem ObjectOutputStream verkettet).

So könnten die Spielfigur-Daten aussehen, wenn Sie sie als von Menschen lesbare Textdatei schreiben würden.

```
50,Elb,Bogen,Schwert,Staub
200,Troll,bloße Hände,große Axt
120,Zauberer,Zaubersprüche,
Unsichtbarkeit
```

### Ein serialisiertes Objekt schreiben:

```
objectOutputStream.writeObject(einObject);
```

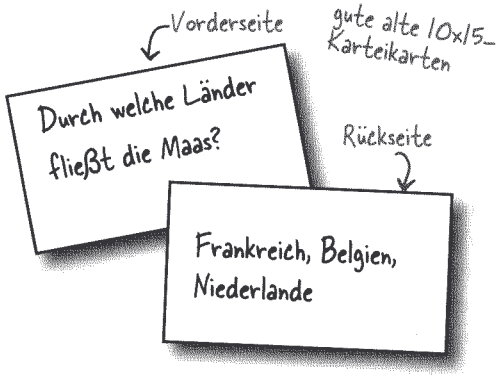
### Einen String schreiben:

```
fileWriter.write("Mein erster zu speichernder String");
```

```
import java.io.*;                          ← Wir brauchen das Paket java.io für den FileWriter.

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

      try {                                          ↓ Wenn die Datei »duda.txt« nicht existiert, wird sie von FileWriter erzeugt.
         FileWriter writer = new FileWriter("duda.txt");

         writer.write("Hallo, du da!");   ← Die Methode write() nimmt einen String entgegen.

         writer.close();                  ← Wieder zumachen, wenn Sie fertig sind!

      } catch(IOException ex) {
        ex.printStackTrace();
      }
   }
}
```

Der gesamte E/A-Kram muss in einem try/catch-Block stehen. Alles kann eine IOException auslösen!!

# Textdatei-Beispiel: E-Lernkarten



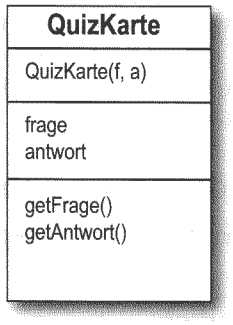

Erinnern Sie sich an diese Lernkarten, die Sie in der Schule benutzt hatten? Wo auf der einen Seite die Frage und auf der Rückseite die Antwort stand? Wenn Sie etwas zu verstehen versuchen, sind sie nicht besonders hilfreich, aber zum Einpauken und Auswendiglernen sind sie unschlagbar. *Wenn Sie Fakten ins Gehirn einbrennen müssen.* Und auch für Ratespiele sind sie prima geeignet.

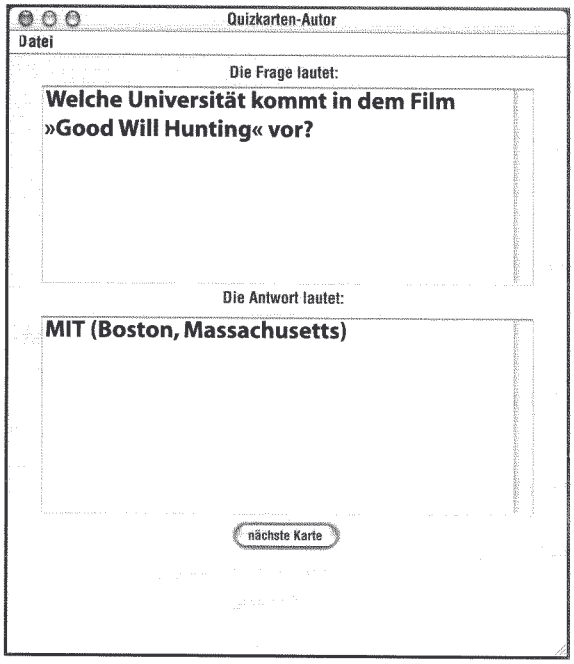
**Wir machen hier eine elektronische Version mit drei Klassen:**

1) ***QuizKartenAutor,*** ein einfaches Tool, mit dem man einen Satz von E-Lernkarten erstellen und speichern kann.

2) ***QuizKartenPlayer,*** ein Abspielgerät, das einen Satz von Lernkarten laden und sie dem Benutzer anzeigen kann.

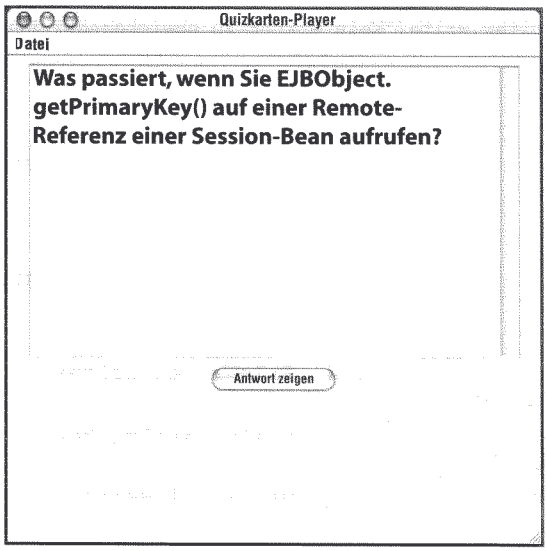
3) ***QuizKarte,*** eine einfache Klasse, die die Daten auf den Karten repräsentiert. Wir werden den Code für das Autorenprogramm und den Player hier durchgehen und Sie die Klasse QuizKarte dann selbst schreiben lassen, und zwar hiermit →




**QuizKartenAutor**

Besitzt ein Menü Datei mit einer Option Speichern, damit der aktuelle Kartensatz in einer Textdatei gespeichert werden kann.




**QuizKartenPlayer**

Besitzt ein Menü Datei mit einer Option Laden, damit ein Kartensatz aus einer Textdatei geladen werden kann.

# QuizKartenAutor (Code grob skizziert)

```
public class QuizKartenAutor {

  public void los() {
    // GUI erstellen und anzeigen
  }

  private class NächsteKarteListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // aktuelle Karte der Liste hinzufügen und Textbereiche leeren
    }
  }

  private class MenüSpeichernListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // ein Dateidialogfeld anzeigen und den Kartensatz vom Benutzer
      // unter einem von ihm gewählten Namen speichern lassen
    }
  }

  private class MenüNeuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // Kartenliste und Textbereiche »abräumen«
    }
  }

  private void dateiSpeichern(File datei) {
    // über die Kartenliste iterieren und alle Karten in einer Weise in eine Textdatei
    // schreiben, dass der Text geparst werden kann (d.h. mit Trennzeichen
    // zwischen den einzelnen Teilen)
  }
}
```

Erstellt die GUI und zeigt sie an, einschließlich Erzeugung und Registrierung der Event-Listener.

Innere Klasse

Wird ausgelöst, wenn der Benutzer auf den Button >>Nächste Karte<< klickt, was bedeutet, dass er diese Karte in der Liste speichern und eine neue Karte haben will.

Innere Klasse

Wird ausgelöst, wenn der Benutzer >>Speichern<< aus dem Menü >>Datei<< auswählt, also alle Karten aus der aktuellen Liste als >>Satz<< speichern möchte (z.B. den Quantenmechanik-Satz, das Hollywood-Quiz, Java-Regeln usw.)

Innere Klasse

Wird ausgelöst, wenn der Benutzer >>Neu<< im Menü >>Datei<< auswählt, d.h. einen nagelneuen Kartensatz haben will (wir räumen daher die Kartenliste und die Textbereiche frei).

Wird vom MenüSpeichernListener aufgerufen; besorgt das eigentliche Schreiben der Datei.

## Code für den QuizKartenAutor

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class QuizKartenAutor {

  private JTextArea frage;
  private JTextArea antwort;
  private ArrayList<QuizKarte> kartenListe;
  private JFrame frame;

  public static void main (String[] args) {
    QuizKartenAutor editor = new QuizKartenAutor();
    editor.los();
  }

  public void los() {
    // GUI erstellen

    frame = new JFrame("Quizkarten-Autor");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel mainPanel = new JPanel();
    Font bigFont = new Font("sanserif", Font.BOLD, 24);
    frage = new JTextArea(6,20);
    frage.setLineWrap(true);
    frage.setWrapStyleWord(true);
    frage.setFont(bigFont);

    JScrollPane fScroller = new JScrollPane(frage);
    fScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
    fScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

    antwort = new JTextArea(6,20);
    antwort.setLineWrap(true);
    antwort.setWrapStyleWord(true);
    antwort.setFont(bigFont);

    JScrollPane aScroller = new JScrollPane(antwort);
    aScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
    aScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

    JButton nächsteKarteButton = new JButton("Nächste Karte");

    kartenListe = new ArrayList<QuizKarte>();

    JLabel fLabel = new JLabel("Die Frage lautet:");
    JLabel aLabel = new JLabel("Die Antwort lautet:");

    mainPanel.add(fLabel);
    mainPanel.add(fScroller);
    mainPanel.add(aLabel);
    mainPanel.add(aScroller);
    mainPanel.add(nächsteKarteButton);
    nächsteKarteButton.addActionListener(new NächsteKarteListener());
    JMenuBar menüleiste = new JMenuBar();
    JMenu menüDatei = new JMenu("Datei");
    JMenuItem menüPunktNeu = new JMenuItem("Neu");
```

Das hier ist alles GUI-Code. Nichts Besonderes, aber vielleicht möchten Sie sich den Code für Menüleiste, Menü und Menüpunkte ansehen.

```
    JMenuItem menüPunktSpeichern = new JMenuItem("Speichern");
    menüPunktNeu.addActionListener(new MenüNeuListener());

    menüPunktSpeichern.addActionListener(new MenüSpeichernListener());
    menüDatei.add(menüPunktNeu);
    menüDatei.add(menüPunktSpeichern);
    menüleiste.add(menüDatei);
    frame.setJMenuBar(menüleiste);
    frame.getContentPane().add(BorderLayout.CENTER, mainPanel);
    frame.setSize(500,600);
    frame.setVisible(true);
  }

  public class NächsteKarteListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {

      QuizKarte karte = new QuizKarte(frage.getText(), antwort.getText());
      kartenListe.add(karte);
      karteAbräumen();
    }
  }

  public class MenüSpeichernListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      QuizKarte karte = new QuizKarte(frage.getText(), antwort.getText());
      kartenListe.add(karte);

      JFileChooser dateiWahl = new JFileChooser();
      dateiWahl.showSaveDialog(frame);
      dateiSpeichern(dateiWahl.getSelectedFile());
    }
  }

  public class MenüNeuListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      kartenListe.clear();
      karteAbräumen();
    }
  }

  private void karteAbräumen() {
    frage.setText("");
    antwort.setText("");
    frage.requestFocus();
  }

  private void dateiSpeichern(File datei) {
    try {
      BufferedWriter writer = new BufferedWriter(new FileWriter(datei));

      for (QuizKarte karte:kartenListe) {
        writer.write(karte.getFrage() + "/");
        writer.write(karte.getAntwort() + "\n");
      }
      writer.close();

    } catch(IOException ex) {
      System.out.println("konnte die Kartenliste nicht schreiben");
      ex.printStackTrace();
    }
  } // Methode schließen
} // Klasse schließen
```

Wir erzeugen eine Menüleiste und ein Menü >>Datei<<, in das wir anschließend die Menüpunkte >>Neu<< und >>Speichern<< setzen. Wir fügen das Menü der Menüleiste hinzu und weisen dann den Frame an, diese zu benutzen. Die Menüpunkte können ein ActionEvent absetzen.

Öffnet ein Dateidialogfeld und bleibt so lange auf dieser Zeile stehen, bis der Benutzer im Dialog auf >>Speichern<< klickt. Die ganze Dateidialognavigation, die Auswahl einer Datei usw. erledigt der JFileChooser für Sie! Ja, es ist tatsächlich so einfach.

Die Methode, die das eigentliche Schreiben in die Datei erledigt (aufgerufen durch den Event-Handler des MenüSpeichernListener). Das Argument ist das >>File<<-Objekt, das der Benutzer speichert. Die Klasse File sehen wir uns auf der nächsten Seite an.

Wir verketten einen BufferedWriter mit einem neuen FileWriter, damit wir effizienter schreiben (darüber sprechen wir ein paar Seiten später noch).

Wir gehen die ArrayList mit den Karten durch und schreiben diese in die Datei, mit einer Karte pro Zeile, Frage und Antwort durch ein >>/<< getrennt, gefolgt von einem Zeilenumbruch (>>\n<<).

# Die Klasse java.io.File

Die Klasse java.io.File *repräsentiert* eine Datei auf einem Datenträger, aber nicht den *Inhalt* der Datei. (Wie bitte?) Stellen Sie sich unter einem File-Objekt eher so etwas vor wie den *Pfadnamen* einer Datei (oder sogar ein Verzeichnis) und nicht die echte Datei höchstpersönlich. Die Klasse File hat beispielsweise keine Methoden zum Lesen und Schreiben. SEHR nützlich daran ist, dass es viel sicherer ist, eine Datei durch ein File-Objekt zu repräsentieren als durch einen String-Dateinamen. Die meisten Klassen, die in ihrem Konstruktor einen Dateinamen als String entgegennehmen (z.B. FileWriter oder FileInputStream), können stattdessen auch ein File-Objekt annehmen. Man kann ein File-Objekt erzeugen, prüfen, ob man einen gültigen Pfad hat, usw. und dann dieses File-Objekt dem FileWriter oder dem FileInputStream übergeben.

### Ein paar Dinge, die Sie mit einem File-Objekt tun können:

① **Ein File-Objekt erzeugen, das eine vorhandene Datei repräsentiert**

```
File f = new File("MeinCode.txt");
```

② **Ein neues Verzeichnis anlegen**

```
File dir = new File("Kapitel7");
dir.mkdir();
```

③ **Den Inhalt eines Verzeichnisses auflisten**

```
if (dir.isDirectory()) {
  String[] dirInhalt = dir.list();
  for (int i = 0; i < dirInhalt.length; i++) {
    System.out.println(dirInhalt[i]);
  }
}
```

④ **Den absoluten Pfad einer Datei oder eines Verzeichnisses erhalten**

```
System.out.println(dir.getAbsolutePath());
```

⑤ **Eine Datei oder ein Verzeichnis löschen (gibt bei Erfolg true zurück)**

```
boolean isDeleted = f.delete();
```

**Ein File-Objekt repräsentiert den Namen und den Pfad einer Datei oder eines Verzeichnisses auf dem Datenträger. Ein Beispiel:**

**/Benutzer/Kati/Daten/SpielDatei.txt**

**Aber es repräsentiert weder die Daten *in* der Datei noch gibt es Ihnen Zugriff darauf!**

Eine Adresse ist NICHT dasselbe wie das tatsächliche Haus! Ein File-Objekt ist wie eine Straßenanschrift ... es repräsentiert den Namen und den Speicherort einer bestimmten Datei, aber es ist nicht die Datei selbst.

Ein File-Objekt repräsentiert den Dateinamen »SpielDatei.txt«.

**SpielDatei.txt**

50,Elb,Bogen,Schwert,Staub

200,Troll,bloße Hände,große Axt

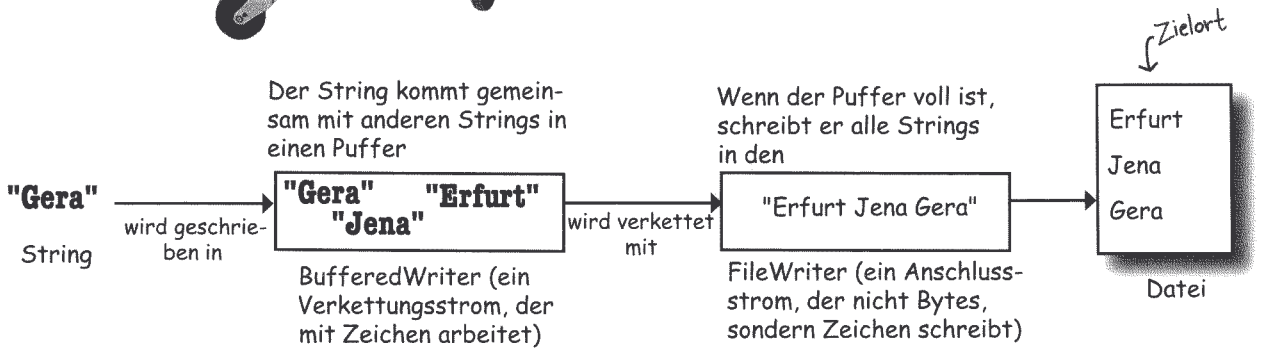
120,Zauberer,Zaubersprüche, Unsichtbarkeit

Ein File-Objekt repräsentiert NICHT die Daten in der Datei (und gibt Ihnen auch keinen Zugriff darauf)!

# Mehr Effizienz mit Puffern

**Wenn es keine Puffer gäbe, wäre das wie Einkaufen ohne Einkaufswagen. Sie müssten alles einzeln zu Ihrem Auto bringen – jede Dosensuppe und jede Toilettenpapierrolle.**

Puffer bieten Ihnen einen vorübergehenden Aufbewahrungsort, an dem Sie Dinge so lange zwischenspeichern können, bis das Behältnis voll ist (wie z.B. der Einkaufswagen). Mit einem Puffer müssen Sie wesentlich weniger Wege zurücklegen.




```
BufferedWriter writer = new BufferedWriter(new FileWriter(eineDatei));
```

Beachten Sie, dass wir noch nicht einmal eine Referenz auf das FileWriter-Objekt halten müssen. Für uns ist nur der BufferedWriter wichtig, denn er ist das Objekt, auf dem wir Methoden aufrufen. Und wenn wir den BufferedWriter schließen, kümmert er sich um den Rest der Kette.

Das Tolle an Puffern ist, dass man mit Ihnen *viel* effizienter arbeiten kann als ohne sie. Man kann auch nur mit Hilfe eines FileWriter in eine Datei schreiben, indem man write(einString) aufruft, aber FileWriter schreibt dann jedes Mal jede winzige Kleinigkeit, die man der Datei übergibt. Das ist ein unerwünschter Overhead, denn jeder Gang zum Datenträger bedeutet – verglichen mit der Manipulation von Daten im Speicher – einen Riesenaufwand. Wenn Sie einen BufferedWriter mit einem FileWriter verketten, nimmt der BufferedWriter all das auf, was Sie schreiben – so lange, bis er voll ist. *Erst wenn der Puffer voll ist, erhält der FileWriter die Anweisung, in die Datei auf dem Datenträger zu schreiben.*

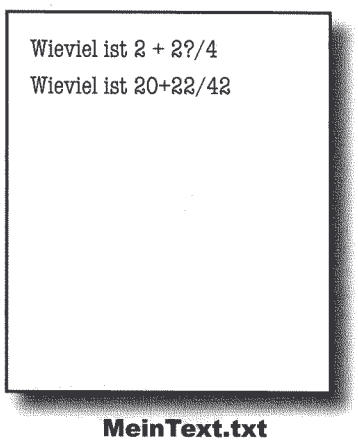
Wenn Sie Daten absenden wollen, *bevor* der Puffer voll ist, können Sie auch dies. Spülen Sie den Puffer einfach mit ***flush()*** leer. Ein Aufruf von writer.flush() sagt: »Sende alles, was im Puffer ist, und zwar ***jetzt!***«

# Aus einer Textdatei lesen

Text aus einer Datei zu lesen ist einfach, aber dieses Mal nehmen wir dafür ein File-Objekt, das die Datei repräsentiert, einen FileReader, der das eigentliche Lesen erledigt, und einen BufferedReader, der das Lesen effizienter macht.

Das Lesen erfolgt über das zeilenweise Einlesen in einer *while*-Schleife, die beendet wird, wenn das Ergebnis von readLine() null ist. Das ist die am meisten verbreitete Art, wie Daten gelesen werden (das gilt praktisch für alles außer serialisierten Objekten): Etwas wird in einer while-Schleife eingelesen (genau genommen in einem while-Schleifen-Prüfvorgang), die abgebrochen wird, wenn nichts mehr zum Lesen da ist (wir merken das daran, dass die von uns verwendete Lesemethode null zurückgibt).

Eine Datei mit zwei Textzeilen.




**MeinText.txt**

Vergessen Sie nicht den Import.

```
import java.io.*;

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

    try {
      File meineDatei = new File("MeinText.txt");
      FileReader fileReader = new FileReader(meineDatei);

      BufferedReader reader = new BufferedReader(fileReader);

      String zeile = null;

      while ((zeile = reader.readLine()) != null) {
        System.out.println(zeile);
      }
      reader.close();

    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Ein FileReader ist ein Anschlussstrom für Zeichen, der sich mit einer Textdatei verbindet.

Verketten Sie den FileReader mit einem BufferedReader, damit Sie effizienter lesen können. Er kehrt erst dann wieder zum Lesen zur Datei zurück, wenn der Puffer leer ist (weil das Programm alles daraus gelesen hat).

Erzeugen Sie eine String-Variable, die jeweils die Zeile aufnimmt, während sie gelesen wird.

Dies besagt: »Lies eine Textzeile und weise sie der String-Variablen namens Zeile zu. Solange diese Variable nicht null ist (weil noch etwas zum Lesen da war), gib die Zeile aus, die gerade gelesen wurde.«

Anders ausgedrückt: »Solange Zeilen zum Lesen da sind, lies sie und gib sie aus.«

# QuizKartenPlayer (Code grob skizziert)

```
public class QuizKartenPlayer {
  public void los() {
    // GUI erstellen und anzeigen
  }

  private class NächsteKarteListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // falls dies eine Frage ist, die Antwort zeigen, sonst nächste Frage zeigen
      // einen Schalter dafür setzen, ob wir gerade eine Frage oder eine Antwort sehen
    }
  }

  private class MenüÖffnenListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // ein Dateidialogfeld anzeigen
      // zu öffnenden Kartensatz mit Hilfe der Navigation vom Benutzer auswählen lassen
    }
  }

  private void dateiLaden(File datei) {
    // muss eine ArrayList von Karten erstellen durch Einlesen aus einer Textdatei
    // wird vom MenüÖffnenListener-EventHandler aufgerufen, liest die Datei zeilenweise
    // und weist die Methode karteErstellen() an, aus der Zeile eine neue Karte zu erzeugen
    // (eine Zeile der Datei enthält sowohl die Frage als auch die Antwort, getrennt durch ein »/«)
  }

  private void karteErstellen(String zuParsendeZeile) {
    // wird durch die Methode dateiLaden() aufgerufen, empfängt eine Zeile aus der Textdatei,
    // zerlegt diese in zwei Teile – Frage und Antwort – und erzeugt daraus eine neue QuizKarte,
    // die sie zu der ArrayList namens kartenListe hinzufügt
  }
}
```

## Code für den QuizKartenPlayer

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class QuizKartenPlayer {

  private JTextArea anzeige;
  private ArrayList<QuizKarte> kartenListe;
  private QuizKarte aktuelleKarte;
  private int aktuelleKarteIndex;
  private JFrame frame;
  private JButton nächsteKarteButton;
  private boolean istAntwortAngezeigt;

  public static void main (String[] args) {
    QuizKartenPlayer reader = new QuizKartenPlayer();
    reader.los();
  }

  public void los() {

    //GUI aufbauen

    frame = new JFrame("Quizkarten-Player");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel hauptPanel = new JPanel();
    Font bigFont = new Font("sanserif", Font.BOLD, 24);

    anzeige = new JTextArea(10,20);
    anzeige.setFont(bigFont);

    anzeige.setLineWrap(true);
    anzeige.setEditable(false);

    JScrollPane fScroller = new JScrollPane(anzeige);
    fScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
    fScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
    nächsteKarteButton = new JButton("Frage zeigen");
    hauptPanel.add(fScroller);
    hauptPanel.add(nächsteKarteButton);
    nächsteKarteButton.addActionListener(new NächsteKarteListener());

    JMenuBar menüleiste = new JMenuBar();
    JMenu menüDatei = new JMenu("Datei");
    JMenuItem menüPunktLaden = new JMenuItem("Kartensatz laden");
    menüPunktLaden.addActionListener(new MenüÖffnenListener());
    menüDatei.add(menüPunktLaden);
    menüleiste.add(menüDatei);
    frame.setJMenuBar(menüleiste);
    frame.getContentPane().add(BorderLayout.CENTER, hauptPanel);
    frame.setSize(500,600);
    frame.setVisible(true);

  } // los schließen
```

Nur GUI-Code auf dieser Seite; nichts Besonderes.

```
public class NächsteKarteListener implements ActionListener {
  public void actionPerformed(ActionEvent ev) {
    if (istAntwortAngezeigt) {
      // die Antwort anzeigen, weil die Frage schon gezeigt wurde
      anzeige.setText(aktuelleKarte.getAntwort());
      nächsteKarteButton.setText("Nächste Karte");
      istAntwortAngezeigt = false;
    } else {
      // nächste Frage zeigen
      if (aktuelleKarteIndex < kartenListe.size()) {
        nächsteKarteZeigen();
      } else {
        // es gibt keine Karten mehr!
        anzeige.setText("Das war die letzte Karte.");
        nächsteKarteButton.setEnabled(false);
      }
    }
  }
}

public class MenüÖffnenListener implements ActionListener {
  public void actionPerformed(ActionEvent ev) {
    JFileChooser dateiÖffnen = new JFileChooser();
    dateiÖffnen.showOpenDialog(frame);
    dateiLaden(dateiÖffnen.getSelectedFile());
  }
}

private void dateiLaden(File datei) {
  kartenListe = new ArrayList<QuizKarte>();
  try {
    BufferedReader reader = new BufferedReader(new FileReader(datei));
    String zeile = null;
    while ((zeile = reader.readLine()) != null) {
      karteErstellen(zeile);
    }
    reader.close();
  } catch(Exception ex) {
    System.out.println("konnte Kartendatei nicht lesen");
    ex.printStackTrace();
  }
// fangen wir an und zeigen die erste Karte:
nächsteKarteZeigen();
}

private void karteErstellen(String zuParsendeZeile) {
  String[] ergebnis = zuParsendeZeile.split("/");
  QuizKarte karte = new QuizKarte(ergebnis[0], ergebnis[1]);
  kartenListe.add(karte);
  System.out.println("Karte erstellt");
}

private void nächsteKarteZeigen() {
  aktuelleKarte = kartenListe.get(aktuelleKarteIndex);
  aktuelleKarteIndex++;
  anzeige.setText(aktuelleKarte.getFrage());
  nächsteKarteButton.setText("Antwort zeigen");
  istAntwortAngezeigt = true;
}
} // Klasse schließen
```

Überprüfen Sie den boolean-Schalter istAntwortAngezeigt, um festzustellen, ob im Moment eine Frage oder eine Antwort angezeigt wird, und tun Sie dann das Passende.

Öffnet das Dateidialogfeld und lässt den Benutzer navigieren und die zu öffnende Datei auswählen.

Machen Sie einen BufferedReader, der mit einem neuen FileReader verkettet ist, und übergeben Sie dem FileReader das File-Objekt, das der Benutzer im Datei öffnen-Dialog auswählt.

Lesen Sie die Datei zeilenweise ein und übergeben Sie die Zeile der Methode karteErstellen, die sie parst, eine echte Quizkarte daraus macht und diese der ArrayList hinzufügt.

Jede Textzeile entspricht einer einzelnen Lernkarte, aber wir müssen die Frage und die Antwort getrennt herausholen. Mit der String-Methode split() spalten wir die Zeile in zwei Tokens (eins für die Frage und eins für die Antwort). Die Methode split() sehen wir uns auf der nächsten Seite an.

# Parsen mit der String-Methode split()

**Stellen Sie sich vor, Sie haben eine solche Lernkarte:**

**Gespeichert in einer Frage-/Antwort-Datei wie dieser:**

```
Was ergibt Blau + Gelb?/Grün
Was ergibt Rot + Blau?/Lila
```

### Wie trennen Sie die Frage von der Antwort?

Wenn Sie die Datei lesen, sehen Sie, dass Frage und Antwort gemeinsam in eine Zeile gequetscht sind, getrennt durch einen Schrägstrich »/« (weil wir die Datei im QuizKartenAutor-Code so geschrieben haben).

### Mit der String-Methode split() können Sie einen String in Stücke zerlegen.

Die Methode split() sagt: »Gib mir ein Feldtrennzeichen, dann breche ich alle in diesem String enthaltenen Stücke für dich auseinander und stecke sie in ein String-Array.«

Im QuizKartenPlayer-Programm sieht eine einzelne Zeile beim Einlesen aus einer Datei so aus.

```
String testeMich = "Was ergibt Blau + Gelb?/Grün";
String[] ergebnis = testeMich.split("/");
for (String token:ergebnis) {
  System.out.println(token);
}
```

Die Methode split() empfängt das >>/<< und zerlegt damit den String in (in diesem Fall) zwei Stücke. (Anmerkung: split() kann VIEL mehr als das, wofür wir es hier verwenden. Sie kann extrem komplexes Parsing mit Filtern, Platzhaltern usw. durchführen.

Durchlaufen Sie das Array mit der Schleife und geben Sie jedes Token (Bruchstück) aus. In diesem Beispiel gibt es nur zwei Tokens: >>Was ergibt Blau + Gelb?<< und >>Grün<<.

## Es gibt keine Dummen Fragen

**F: Also, ich schau in die API, und da gibt es ungefähr 5 Millionen Klassen im Paket java.io. Wie zum Teufel soll man da wissen, welche man nehmen soll?**

A: Die E/A-API nutzt das Konzept der modularen »Verkettung«, so dass Sie Anschlussströme und Verkettungsströme (die auch als Filterströme bezeichnet werden) in einer Vielzahl von Kombinationen zusammenstecken können. So können Sie fast alles bekommen, was Ihr Herz begehrt.

Die Ketten müssen nicht bei zwei Stufen enden. Sie können sich auch Mehrfach-Verkettungsströme zusammenstecken, um genau die Menge an Verarbeitung zu bekommen, die Sie brauchen.

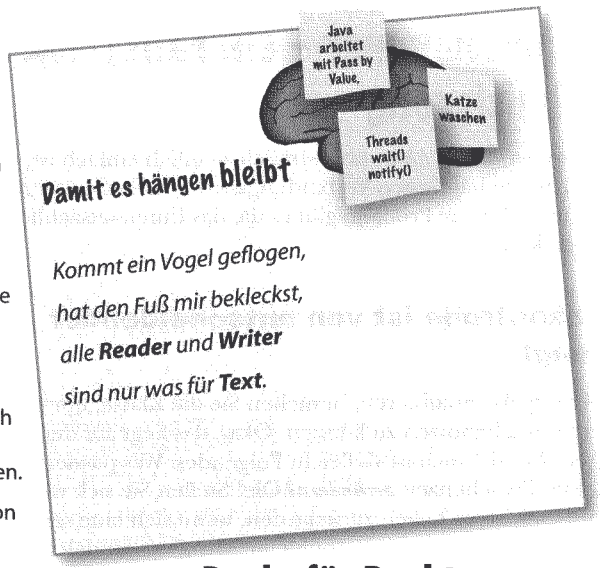
Meistens werden Sie jedoch immer dieselbe kleine Auswahl von Klassen benutzen. Wenn Sie Textdateien schreiben, brauchen Sie wahrscheinlich nur BufferedReader und BufferedWriter (verkettet mit FileReader und FileWriter). Wenn Sie serialisierte Objekte schreiben, können Sie ObjectOutputStream und ObjectInputStream benutzen (verkettet mit FileOutputStream und FileInputStream).

Anders ausgedrückt: 90% dessen, was Sie typischerweise mit Java-E/A machen, lassen sich mit den hier besprochenen Klassen bewältigen.

**F: Was ist mit den neuen nio-E/A-Klassen, die in 1.4 hinzugekommen sind?**

A: Die java.nio-Klassen sind wesentlich leistungsfähiger und nutzen die nativen Möglichkeiten des Rechners, auf dem Ihr Programm läuft, besser aus. Eine der wichtigsten neuen Eigenschaften von NIO ist, dass Sie eine direkte Kontrolle über Puffer haben. Ein weiteres neues Merkmal ist nicht blockierende E/A, was bedeutet, dass Ihr E/A-Code nicht einfach dasitzt und wartet, wenn es nichts zum Lesen oder Schreiben gibt. Ein paar der älteren Klassen (einschließlich FileInputStream und FileOutputStream) nutzen unter der Oberfläche einige der neuen Möglichkeiten. Die NIO-Klassen sind allerdings schwieriger zu verwenden; wenn Sie also die neuen Funktionen nicht *wirklich* brauchen, sollten Sie vielleicht besser bei den einfacheren Versionen bleiben, die wir hier benutzt haben. Außerdem kann NIO, wenn Sie nicht aufpassen, zu einem Leistungs*verlust* führen. Wahrscheinlich wird für 90% des Codes, den Sie schreiben werden, Nicht-NIO-E/A gut geeignet sein, besonders wenn Sie gerade erst mit Java anfangen.

Aber Sie können sich schon einmal an die nio-Klassen herantasten, indem Sie FileInputStream verwenden und über die (mit Version 1.4 zum FileInputStream hinzugefügte) Methode getChannel() auf seinen *Channel* zugreifen.




## Punkt für Punkt

- Um eine Textdatei zu schreiben, beginnen Sie mit einem FileWriter-Anschlussstrom.
- Verketten Sie den FileWriter zwecks besserer Effizienz mit einem BufferedWriter.
- Ein File-Objekt repräsentiert eine Datei mit einem bestimmten Pfad, jedoch nicht den tatsächlichen Inhalt der Datei.
- Mit einem File-Objekt können Sie Verzeichnisse erzeugen, durchlaufen und löschen.
- Die meisten Ströme, die einen String-Dateinamen benutzen können, können stattdessen auch ein File-Objekt benutzen; Letzteres kann sicherer sein.
- Um eine Textdatei zu lesen, beginnen Sie mit einem FileReader-Anschlussstrom.
- Verketten Sie den FileReader zwecks besserer Effizienz mit einem BufferedReader.
- Damit eine Textdatei geparst werden kann, muss sie in einer Weise geschrieben worden sein, dass man die verschiedenen Elemente wiedererkennen kann. Häufig wird dies so gelöst, dass man die einzelnen Stücke durch ein bestimmtes Zeichen trennt.
- Spalten Sie einen String mit der String-Methode split() in einzelne Tokens auf. Ein String mit nur *einem* Feldtrennzeichen enthält zwei Tokens, eins auf jeder Seite des Trennzeichens. *Das Trennzeichen selbst zählt nicht als Token.*

# Die Versionsnummer – ein Fallstrick bei der Serialisierung

Sie haben gesehen, dass E/A in Java wirklich ziemlich einfach ist, vor allem wenn Sie sich an die häufigsten Verbindungsstrom-Verkettungsstrom-Kombinationen halten. Aber *ein* Problem gibt es da, das Ihnen tatsächlich Schwierigkeiten machen könnte.

## Versionskontrolle ist von entscheidender Bedeutung!

Wenn Sie ein Objekt serialisieren, brauchen Sie die Klasse, um es wieder deserialisieren und benutzen zu können. Okay, das liegt auf der Hand. Aber weniger offensichtlich ist vielleicht Folgendes: Was passiert, wenn Sie ***die Klasse*** in der Zwischenzeit ***verändern?*** Oje! Stellen Sie sich vor, Sie wollen ein Hund-Objekt ins Leben zurückrufen, wenn sich eine seiner (nicht transienten) Instanzvariablen von double in String verwandelt hat. Das ist ein heftiger Verstoß gegen die Typsicherheit in Java. Doch das ist nicht die einzige Veränderung, die die Kompatibilität beeinträchtigen könnte. Denken Sie mal an Folgendes:

### Veränderungen an einer Klasse, die bei der Deserialisierung schaden könnten:

Eine Instanzvariable löschen

Den deklarierten Typ einer Instanzvariablen ändern

Eine nicht transiente Instanzvariable transient machen

Eine Klasse in der Vererbungshierarchie nach oben oder nach unten verschieben

Aus einer serialisierbaren Klasse (irgendwo innerhalb des Objektgraphen) eine nicht serialisierbare machen (durch Löschen von »implements Serializable« in einer Klassendeklaration)

Eine Instanzvariable statisch machen

### Veränderungen an einer Klasse, die in der Regel okay sind:

Neue Instanzvariablen zu einer Klasse hinzufügen (vorhandene Objekte erhalten bei der Initialisierung Standardwerte für diejenigen Instanzvariablen, die sie bei ihrer Serialisierung noch nicht besaßen)

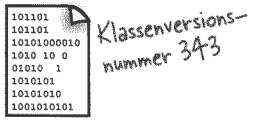
Klassen zum Vererbungsbaum hinzufügen

Klassen aus dem Vererbungsbaum entfernen

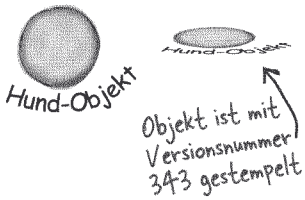
Die Sichtbarkeitsmodifizierer einer Instanzvariablen ändern. Der Variablen kann bei der Deserialisierung trotzdem auf jeden Fall ein Wert zugewiesen werden.

Eine Instanzvariable von transient in nicht transient umwandeln (vorher serialisierten Objekten wird einfach ein Standardwert für die ehemals transienten Variablen zugewiesen)

1. Sie schreiben eine Klasse Hund.

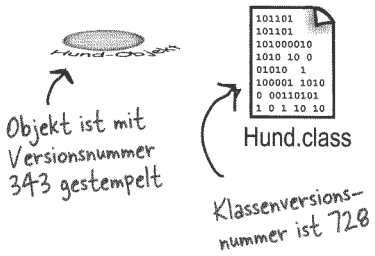



2. Sie serialisieren ein Hund-Objekt unter Verwendung dieser Klasse.




3. Sie verändern die Klasse Hund.

4. Sie deserialisieren ein Hund-Objekt unter Verwendung der veränderten Klasse.




5. Die Deserialisierung schlägt fehl!!

   Die JVM sagt: »Sie können einem alten Hund keinen neuen Code beibringen.«

# Verwendung der serialVersionUID

Jedes Mal, wenn ein Objekt serialisiert wird, wird das Objekt (einschließlich jedes Objekts im Graphen) mit einer Versionsnummer für die Klasse des Objekts »gestempelt«. Diese Nummer bzw. ID bezeichnet man als serialVersionUID (UID steht für unique identifier – also eindeutige Identifikationsnummer); sie wird auf der Basis von Informationen über die Klassenstruktur berechnet. Wenn ein Objekt deserialisiert wird und sich die Klasse seit seiner Serialisierung geändert hat, könnte die Klasse eine andere serialVersionUID haben, und dann schlägt die Deserialisierung fehl! Aber darauf können Sie Einfluss nehmen.

### Wenn Sie es *irgendwie* für möglich halten, dass sich Ihre Klasse *weiterentwickelt*, geben Sie ihr eine serialVersionUID

Wenn Java ein Objekt zu deserialisieren versucht, vergleicht es die serialVersionUID des serialisierten Objekts mit der der Klasse, die von der JVM zur Deserialisierung dieses Objekts verwendet wird. Wurde beispielsweise eine Hund-Instanz mit einer ID von, sagen wir, 23 serialisiert (in der Realität ist eine serialVersionUID wesentlich länger), vergleicht die JVM bei der Deserialisierung als Erstes die serialVersionUID des Hund-Objekts mit der der Hund-Klasse. Wenn die beiden Zahlen nicht übereinstimmen, geht die JVM davon aus, dass die Klasse nicht mit dem zu einem früheren Zeitpunkt serialisierten Objekt kompatibel ist, und Sie erhalten während der Deserialisierung eine Exception.

Die Lösung besteht darin, eine feste serialVersionUID in Ihre Klasse zu schreiben. Wenn sich dann die Klasse weiterentwickelt, bleibt die serialVersionUID gleich, und die JVM sagt: »Okay, cool, die Klasse ist mit diesem serialisierten Objekt kompatibel« –, obwohl sich die Klasse in Wirklichkeit doch geändert hat.

Das funktioniert *nur dann*, wenn Sie vorsichtig mit den Veränderungen an Ihrer Klasse sind! Mit anderen Worten: *Sie* übernehmen die Verantwortung für alle Probleme, die auftreten, wenn ein älteres Objekt mit einer neuen Klasse wieder zum Leben erweckt wird.

Um eine serialVersionUID für eine Klasse zu erhalten, verwenden Sie das serialver-Tool, das in Ihrem Java-Development-Kit enthalten ist.

```
Datei Bearbeiten Fenster Hilfe serienKiller
% serialver Hund
Hund: static final long
serialVersionUID = -
5849794470654667210L;
```

### Wenn Sie glauben, dass sich Ihre Klasse *weiterentwickelt*, wenn jemand Objekte davon serialisiert hat ...

1. Verwenden Sie das serialver-Kommandozeilen-Tool, um die Versions-ID für Ihre Klasse zu bekommen.

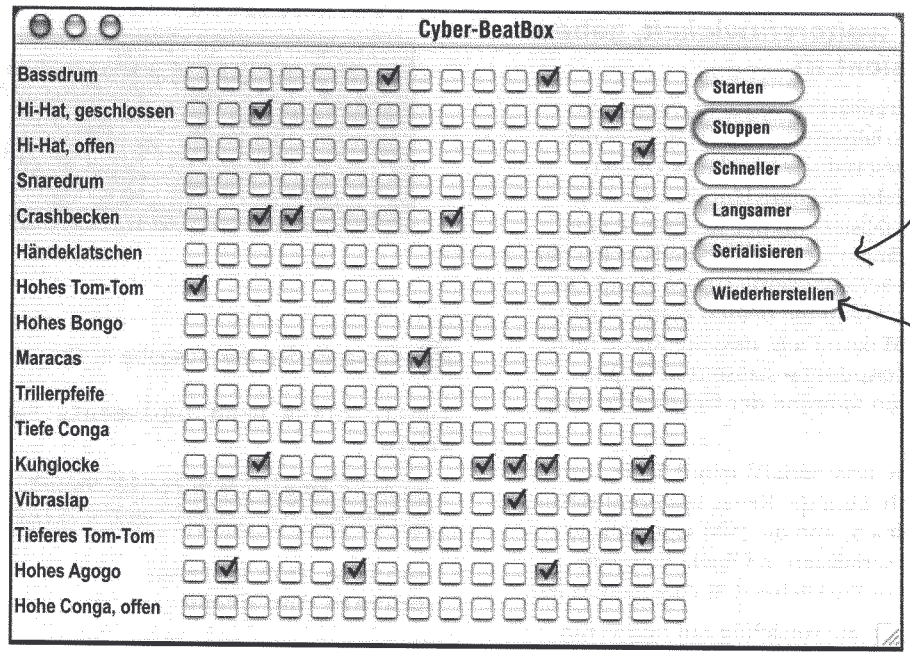
```
Datei Bearbeiten Fenster Hilfe serienKiller
% serialver Hund
Hund: static final long
serialVersionUID = -
5849794470654667210L;
```

2. Kopieren Sie die Ausgabe in Ihre Klasse.

```
public class Hund implements Seriali-
zable {

  static final long serialVersionUID =
                  -5849794470654667210L;

  private String name;
  private int größe;

  // hier Methoden-Code
}
```

3. Wenn Sie Ihre Klasse verändern, müssen Sie sich darüber klar sein, dass Sie mit Ihrem Code die Verantwortung für die Konsequenzen übernehmen, die Ihre Veränderungen an der Klasse haben könnten! Sorgen Sie z.B. dafür, dass Ihre neue Hund-Klasse damit zurechtkommt, wenn ein alter Hund bei der Deserialisierung Standardwerte für Instanzvariablen erhält, die erst *nach* der Serialisierung des Hundes zur Klasse hinzugefügt wurden.

# CodeKüche




**Lassen wir die BeatBox unser Lieblingspattern speichern und wiederherstellen.**

# Speichern eines BeatBox-Patterns

Erinnern Sie sich, in der BeatBox ist ein Drum-Pattern nicht mehr als ein Haufen von Kontrollkästchen. Wenn die Sequenz abgespielt werden soll, geht der Code die Kontrollkästchen durch, um herauszufinden, welche Schlagzeugklänge bei jedem der 16 Schläge zu spielen sind. Um ein Pattern zu speichern, brauchen wir daher nichts weiter zu tun, als den Zustand der Kontrollkästchen zu speichern.

Wir können ein einfaches boolean-Array erzeugen, das den Zustand aller 256 Kontrollkästchen enthält. Ein Array-Objekt ist serialisierbar, solange die Dinge *im* Array serialisierbar sind, daher werden wir keine Probleme haben, ein Array mit boolean-Werten zu speichern.

Um ein Pattern wieder zu laden, lesen wir nur das boolean-Array-Objekt ein (deserialisieren es) und stellen die Kontrollkästchen wieder her. Den größten Teil des Codes haben Sie schon gesehen – in der CodeKüche, in der wir die BeatBox-Benutzeroberfläche gebaut haben. In diesem Kapitel sehen wir uns daher nur den Code zum Speichern und Wiederherstellen an.

Diese CodeKüche bereitet uns auf das nächste Kapitel vor, in dem wir das Muster dann nicht mehr in eine *Datei* schreiben werden, sondern es über das *Netz* an den Server senden. Und statt ein Pattern aus einer Datei zu laden, erhalten wir Patterns vom *Server* – jedes Mal, wenn ein Teilnehmer eines an den Server sendet.

### Serialisierung eines Patterns

Dies ist eine innere Klasse innerhalb des BeatBox-Codes.

```
public class MeinSendenListener implements ActionListener {

  public void actionPerformed(ActionEvent a) {

    boolean[] checkboxZustand = new boolean[256];

    for (int i = 0; i < 256; i++) {

      JCheckBox check = (JCheckBox) checkboxListe.get(i);
      if (check.isSelected()) {
        checkboxZustand[i] = true;
      }
    }

    try {
      FileOutputStream fileStream = new FileOutputStream(new File("Checkbox.ser"));
      ObjectOutputStream os = new ObjectOutputStream(fileStream);
      os.writeObject(checkboxZustand);
    } catch(Exception ex) {
      ex.printStackTrace();
    }

  } // Methode schließen
} // innere Klasse schließen
```

Das Ganze passiert, wenn der Benutzer auf den Button klickt und das ActionEvent abgesetzt wird.

Erzeugen Sie ein boolean-Array, das den Zustand jeder Checkbox enthält.

Gehen Sie die checkboxListe (eine ArrayList von Kontrollkästchen) durch, holen Sie sich für jede einzelne den Zustand und fügen Sie ihn dem boolean-Array hinzu.

Dieser Teil ist ein Kinderspiel. Einfach nur das boolean-Array schreiben/serialisieren!

# Wiederherstellung eines BeatBox-Patterns

Dies ist so ungefähr das Gleiche rückwärts ... das boolean-Array wird gelesen und dazu verwendet, den Zustand der GUI- Kontrollkästchen wiederherzustellen. Das Ganze passiert, wenn der Benutzer auf den Button »Wiederherstellen« drückt.

## Pattern wiederherstellen

Dies ist eine weitere innere Klasse innerhalb der BeatBox-Klasse.

```
public class MeinEinlesenListener implements ActionListener {

  public void actionPerformed(ActionEvent a) {
    boolean[] checkboxZustand = null;
    try {
      FileInputStream fileIn = new FileInputStream(new File("Checkbox.ser"));
      ObjectInputStream is = new ObjectInputStream(fileIn);
      checkboxZustand = (boolean[]) is.readObject();

    } catch(Exception ex) {ex.printStackTrace();}


    for (int i = 0; i < 256; i++) {
      JCheckBox check = (JCheckBox) checkboxListe.get(i);
      if (checkboxZustand[i]) {
        check.setSelected(true);
      } else {
        check.setSelected(false);
      }
    }

    sequencer.stop();
    trackErstellenUndStarten();

  } // Methode schließen
} // innere Klasse schließen
```

Lesen Sie das einzige Objekt in der Datei (das boolean-Array) und führen Sie einen Cast auf ein boolean-Array durch (wie Sie wissen, gibt readObject() eine Referenz vom Typ Object zurück).

Stellen Sie jetzt den Zustand der einzelnen Kontrollkästchen in der ArrayList mit den eigentlichen Checkbox-Objekten (checkbox-Liste) wieder her.

Stoppen Sie nun das, was gerade läuft, und bauen Sie die Sequence mit Hilfe der neuen Checkbox-Zustände in der ArrayList neu auf.

### Spitzen Sie Ihren Bleistift

Diese Version hat einen großen Nachteil! Wenn Sie auf den Button »Serialisieren« klicken, wird automatisch in eine Datei namens »Checkbox.ser« serialisiert (falls diese noch nicht existiert, wird sie erzeugt). Doch jedes Mal, wenn Sie speichern, überschreiben Sie die zuvor gespeicherte Datei.

Verbessern Sie die Speichern- und Wiederherstellen-Funktion durch den Einbau eines JFileChoosers, so dass Sie so viele verschiedene Patterns unter verschiedenen Namen speichern können, wie Sie möchten, und *jedes* der zuvor gespeicherten Patterns nach Belieben laden/wiederherstellen können.

Spitzen Sie Ihren Bleistift

## Können sie gespeichert werden?

Welche davon sind Ihrer Ansicht nach serialisierbar oder sollten es sein? Wenn nicht, warum nicht? Nicht sinnvoll? Sicherheitsrisiko? Funktioniert nur bei der aktuellen Ausführung der JVM? Versuchen Sie, es durch Überlegen herauszufinden, ohne in der API nachzuschauen.

| Objekttyp | Serialisierbar? | Falls nicht, warum nicht? |
|---|---|---|
| Object | ja/nein | _____ |
| String | ja/nein | _____ |
| File | ja/nein | _____ |
| Date | ja/nein | _____ |
| OutputStream | ja/nein | _____ |
| JFrame | ja/nein | _____ |
| Integer | ja/nein | _____ |
| System | ja/nein | _____ |

## Was ist erlaubt?

Markieren Sie die Codefragmente, die kompilierbar wären (vorausgesetzt, sie stünden in einer Klasse mit zulässigem Code).

```
FileReader fileReader = new FileReader();
BufferedReader reader = new BufferedReader(fileReader);
```

---

```
FileOutputStream f = new FileOutputStream(new File("Foo.ser"));
ObjectOutputStream os = new ObjectOutputStream(f);
```

---

```
BufferedReader reader = new BufferedReader(new FileReader(file));
String zeile = null;
while ((zeile = reader.readLine()) != null) {
  karteErzeugen(zeile);
}
```

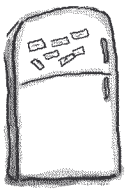
---

```
ObjectInputStream is = new ObjectInputStream(new FileOutputStream("Spiel.ser"));
Spielfigur eins = (Spielfigur) is.readObject();
```

In diesem Kapitel haben wir die wunderbare Welt von Java-E/A erkundet. Entscheiden Sie, ob die folgenden Aussagen dazu jeweils wahr oder falsch sind.

# Wahr oder Falsch?

1. Serialisierung eignet sich, um Daten für die Verwendung durch Nicht-Java-Programme zu speichern.
2. Objektzustände können nur mittels Serialisierung gespeichert werden.
3. Mit der Klasse ObjectOutputStream werden serialisierte Objekte gespeichert.
4. Verkettungsströme können für sich allein oder zusammen mit Anschlussströmen benutzt werden.
5. Ein einziger Aufruf von writeObject() kann die Speicherung von vielen Objekten bewirken.
6. Alle Klassen sind standardmäßig serialisierbar.
7. Mit einem transient-Modifizierer kann man Instanzvariablen serialisierbar machen.
8. Wenn eine Superklasse nicht serialisierbar ist, kann die Unterklasse es auch nicht sein.
9. Bei der Deserialisierung werden Objekte in umgekehrter Reihenfolge, d.h. nach dem Prinzip »last in first out«, eingelesen.
10. Bei der Deserialisierung eines Objekts wird der Konstruktor nicht ausgeführt.
11. Sowohl bei der Serialisierung als auch bei der Speicherung in eine Textdatei können Exceptions ausgelöst werden.
12. Ein BufferedWriter kann mit einem FileWriter verkettet werden.
13. File-Objekte repräsentieren Dateien, aber keine Verzeichnisse.
14. Sie können einen Puffer nicht zum Senden seiner Daten zwingen, bevor er voll ist.
15. Sowohl FileReader als auch FileWriter können wahlweise auch gepuffert werden.
16. Die String-Methode split() nimmt Trennzeichen als Token mit in das zurückgegebene Array auf.
17. *Jede* Veränderung an einer Klasse macht alle zuvor serialisierten Objekte der Klasse unbrauchbar.



# Code-Magneten

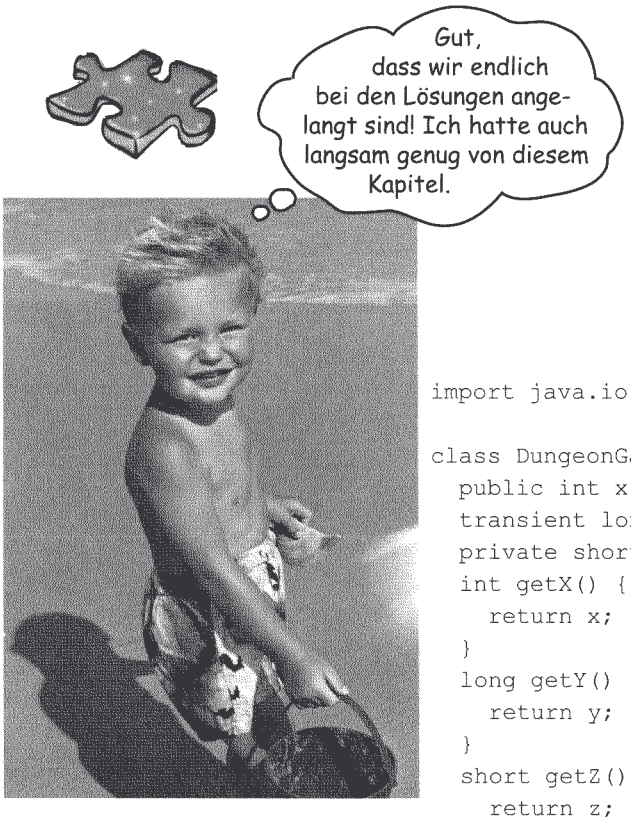
Das hier ist knifflig, deshalb haben wir statt einer Übung ein vollwertiges Puzzle daraus gemacht. Können Sie alle Codeschnipsel wieder zu einem funktionierenden Java-Programm zusammensetzen, das die unten abgedruckte Ausgabe erzeugt? (Sie brauchen möglicherweise nicht alle Magneten, und Sie dürfen einen Magneten auch mehrmals verwenden.)

```
class DungeonGame implements Serializable {

try {

FileOutputStream fos = new
  FileOutputStream("dg.ser");

short getZ() {
  return z;

e.printStackTrace();

oos.close();

ObjectInputStream ois = new
  ObjectInputStream(fis);

int getX() {
  return x;

System.out.println(d.getX()+d.getY()+d.getZ());

FileInputStream fis = new
  FileInputStream("dg.ser");

public int x = 3;
transient long y = 4;
private short z = 5;

long getY() {
  return y;

class DungeonTest {

ois.close();

import java.io.*;

fos.writeObject(d);

} catch (Exception e) {

d = (DungeonGame) ois.readObject();

ObjectOutputStream oos = new
  ObjectOutputStream(fos);

oos.writeObject(d);

public static void main(String [] args) {
  DungeonGame d = new DungeonGame();
```

```
Datei Bearbeiten Fenster Hilfe schnellRausHier
% java DungeonTest
12
8
```

# Lösungen zu den Übungen

1. Serialisierung eignet sich, um Daten für die Verwendung durch Nicht-Java-Programme zu speichern. **Falsch**
2. Objektzustände können nur mittels Serialisierung gespeichert werden. **Falsch**
3. Mit der Klasse ObjectOutputStream werden serialisierte Objekte gespeichert. **Wahr**
4. Verkettungsströme können für sich allein oder zusammen mit Anschlussströmen benutzt werden. **Falsch**
5. Ein einziger Aufruf von writeObject() kann die Speicherung von vielen Objekten bewirken. **Wahr**
6. Alle Klassen sind standardmäßig serialisierbar. **Falsch**
7. Mit einem transient-Modifizierer kann man Instanzvariablen serialisierbar machen. **Falsch**
8. Wenn eine Superklasse nicht serialisierbar ist, kann die Unterklasse es auch nicht sein. **Falsch**
9. Bei der Deserialisierung werden Objekte in umgekehrter Reihenfolge, d.h. nach dem Prinzip »last in first out«, eingelesen. **Falsch**
10. Bei der Deserialisierung eines Objekts wird der Konstruktor nicht ausgeführt. **Wahr**
11. Sowohl bei der Serialisierung als auch bei der Speicherung in eine Textdatei können Exceptions ausgelöst werden. **Wahr**
12. Ein BufferedWriter kann mit einem FileWriter verkettet werden. **Wahr**
13. File-Objekte repräsentieren Dateien, aber keine Verzeichnisse. **Falsch**
14. Sie können einen Puffer nicht zum Senden seiner Daten zwingen, bevor er voll ist. **Falsch**
15. Sowohl FileReader als auch FileWriter können wahlweise auch gepuffert werden. **Wahr**
16. Die String-Methode split() nimmt Trennzeichen als Token mit in das zurückgegebene Array auf. **Falsch**
17. *Jede* Veränderung an einer Klasse macht alle zuvor serialisierten Objekte der Klasse unbrauchbar. **Falsch**



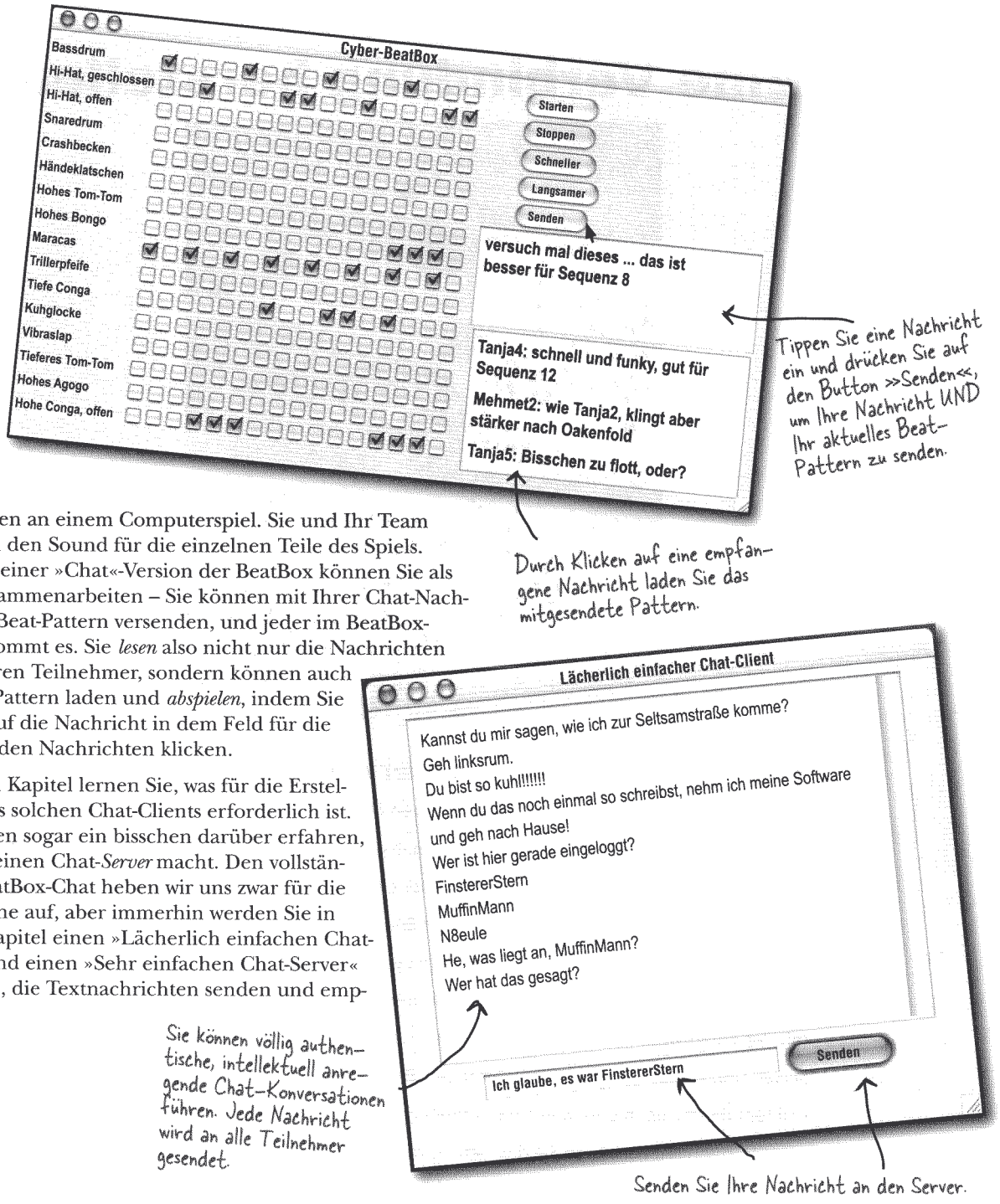

```
import java.io.*;

class DungeonGame implements Serializable {
  public int x = 3;
  transient long y = 4;
  private short z = 5;
  int getX() {
    return x;
  }
  long getY() {
    return y;
  }
  short getZ() {
    return z;
  }
}

class DungeonTest {
  public static void main(String [] args) {
    DungeonGame d = new DungeonGame();
    System.out.println(d.getX() + d.getY() + d.getZ());
    try {
      FileOutputStream fos = new FileOutputStream("dg.ser");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(d);
      oos.close();
      FileInputStream fis = new FileInputStream("dg.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      d = (DungeonGame) ois.readObject();
      ois.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
    System.out.println(d.getX() + d.getY() + d.getZ());
  }
}
```

```
Datei Bearbeiten Fenster Hilfe schnellWegHier
% java DungeonTest
12
8
```

# Eine Verbindung herstellen

**Nehmen Sie Verbindung mit der Außenwelt auf.** Ihr Java-Programm kann in Kontakt mit einem Programm auf einer anderen Maschine treten. Das ist nicht schwer. Um all die Low-level-Netzwerk-Details kümmern sich Klassen in der java.net-Bibliothek. Einer der großen Vorteile von Java ist, dass das Senden und Empfangen über ein Netz einfache Eingabe/Ausgabe ist, lediglich mit einem etwas anderen Verbindungsstrom am Ende der Kette. Wenn Sie einen BufferedReader haben, können Sie *lesen.* Und dem BufferedReader ist es völlig egal, ob die Daten aus einer Datei stammen oder über ein Ethernet-Kabel angeflossen kommen. In diesem Kapitel verbinden wir uns über Sockets mit der Außenwelt. Wir machen *Client*-Sockets. Wir machen *Server*-Sockets. Wir machen *Clients* und *Server.* Und wir lassen sie miteinander reden. Bevor Sie mit dem Kapitel durch sind, haben Sie einen voll funktionsfähigen und Multithread-tauglichen Chat-Client. Haben wir da gerade *Multithread* gesagt? Ja, denn jetzt erfahren Sie das Geheimnis, wie Sie es machen, gleichzeitig mit Thomas zu reden und Anke zuzuhören.

# BeatBox mit Echtzeit-Chat



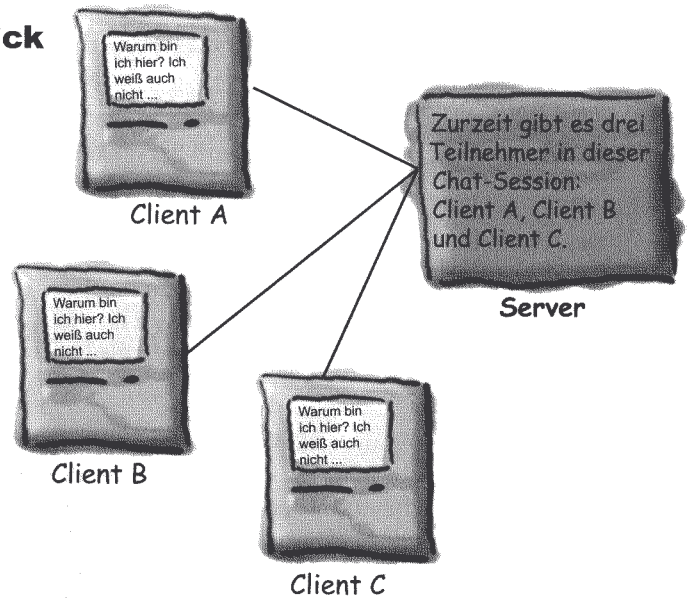

Sie arbeiten an einem Computerspiel. Sie und Ihr Team schreiben den Sound für die einzelnen Teile des Spiels. Mit Hilfe einer »Chat«-Version der BeatBox können Sie als Team zusammenarbeiten – Sie können mit Ihrer Chat-Nachricht ein Beat-Pattern versenden, und jeder im BeatBox-Chat bekommt es. Sie *lesen* also nicht nur die Nachrichten der anderen Teilnehmer, sondern können auch ein Beat-Pattern laden und *abspielen*, indem Sie einfach auf die Nachricht in dem Feld für die eingehenden Nachrichten klicken.

In diesem Kapitel lernen Sie, was für die Erstellung eines solchen Chat-Clients erforderlich ist. Wir werden sogar ein bisschen darüber erfahren, wie man einen Chat-*Server* macht. Den vollständigen BeatBox-Chat heben wir uns zwar für die CodeKüche auf, aber immerhin werden Sie in diesem Kapitel einen »Lächerlich einfachen Chat-Client« und einen »Sehr einfachen Chat-Server« schreiben, die Textnachrichten senden und empfangen.

## Das Chat-Programm im Überblick

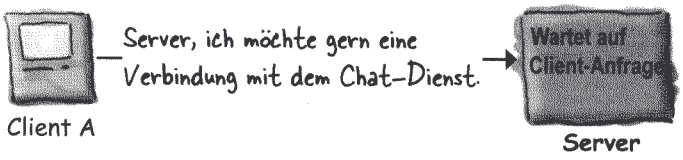
Der Client muss den Server kennen.

Der Server muss ALLE Clients kennen.

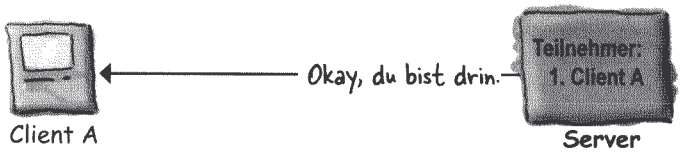



## Wie es funktioniert:

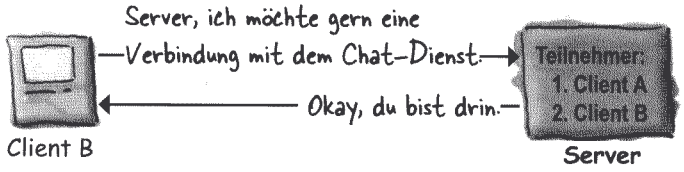
1. Client verbindet sich mit dem Server.



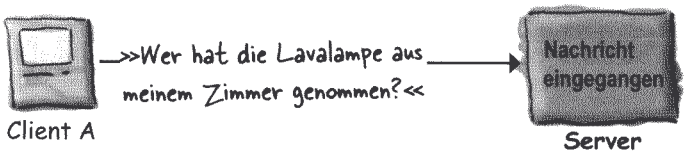

2. Der Server stellt eine Verbindung her und fügt den Client der Liste der Teilnehmer hinzu.



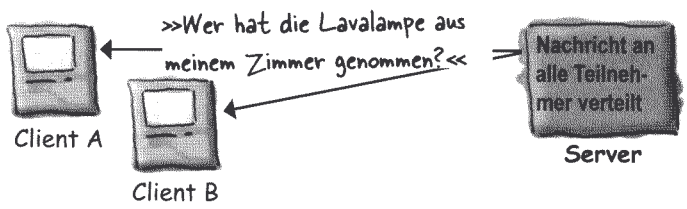

3. Ein anderer Client verbindet sich.




4. Client A sendet eine Nachricht an den Chat-Dienst.




5. Der Server verteilt die Nachricht an ALLE Teilnehmer (einschließlich des ursprünglichen Absenders).



# Verbinden, senden und empfangen

Drei Dinge müssen wir lernen, um den Client zum Laufen zu bringen:

1) Wie man die **Verbindung** zwischen Client und Server herstellt.

2) Wie man Nachrichten *an* den Server **sendet.**

3) Wie man Nachrichten *vom* Server **empfängt.**

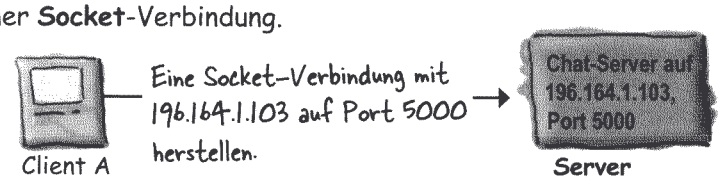
Es muss eine Menge Lowlevel-Kram passieren, damit das Ganze funktioniert. Aber wir haben Glück – das Netzwerkpaket der Java-API macht das zu einem Kinderspiel für Programmierer. Sie werden viel mehr GUI-Code zu sehen bekommen als Netzwerk- und E/A-Code.

Aber das ist noch nicht alles.

In dem einfachen Chat-Client lauert nämlich ein Problem, dem wir in diesem Buch bis jetzt nicht begegnet sind: Wie macht man zwei Dinge gleichzeitig? Das Herstellen einer Verbindung ist eine einzelne Operation (die entweder funktioniert oder scheitert). Aber anschließend will ein Chat-Teilnehmer *ausgehende Nachrichten versenden* und **gleichzeitig** *eingehende Nachrichten* von den anderen Teilnehmern *empfangen* (über den Server). Hmmm ... darüber müssen wir ein bisschen nachdenken; wir kommen ein paar Seiten später darauf zurück.

## ❶ Verbinden

Der Client verbindet sich mit dem Server durch den Aufbau einer **Socket**-Verbindung.




## ❷ Senden

Der Client **sendet** eine Nachricht an den Server.

## ❸ Empfangen

Der Client **erhält** eine Nachricht vom Server.

# Eine Netzwerkverbindung über einen Socket herstellen

Um die Verbindung zu einer anderen Maschine herzustellen, brauchen wir eine Socket-Verbindung. Ein Socket (Klasse: java.net.Socket) ist ein Objekt, das eine Netzwerkverbindung zwischen zwei Maschinen repräsentiert. Eine Verbindung – das ist eine *Beziehung* zwischen zwei Maschinen, bei der **zwei Softwareteile voneinander wissen.** Am wichtigsten dabei ist, dass diese beiden Softwareteile wissen, wie sie miteinander *kommunizieren* können oder anders ausgedrückt, wie sie sich gegenseitig *Bits* zusenden.

Die Lowlevel-Details interessieren uns glücklicherweise nicht, da sie auf einer viel tieferen Stufe im »Netzwerk-Stack« behandelt werden. Falls Sie nicht wissen, was der »Netzwerk-Stack« ist – kein Problem. Das ist einfach eine Art, die verschiedenen Stufen zu betrachten, über die Informationen (Bits) wandern müssen, um von einem Java-Programm, das in einer JVM auf irgendeinem Betriebssystem läuft, in die Hardware (z.B. Ethernet-Kabel) zu gelangen und von dort wieder zurück auf irgendeine andere Maschine. *Irgendjemand* muss sich um all die schmutzigen Details kümmern. Aber dieser Jemand sind nicht Sie, sondern eine Kombination aus betriebssystemspezifischer Software und der Java-Netzwerk-API. Der für Sie wichtige Teil ist »highlevel« – eigentlich *sehr* »highlevel« – und unglaublich einfach. Sind Sie bereit?

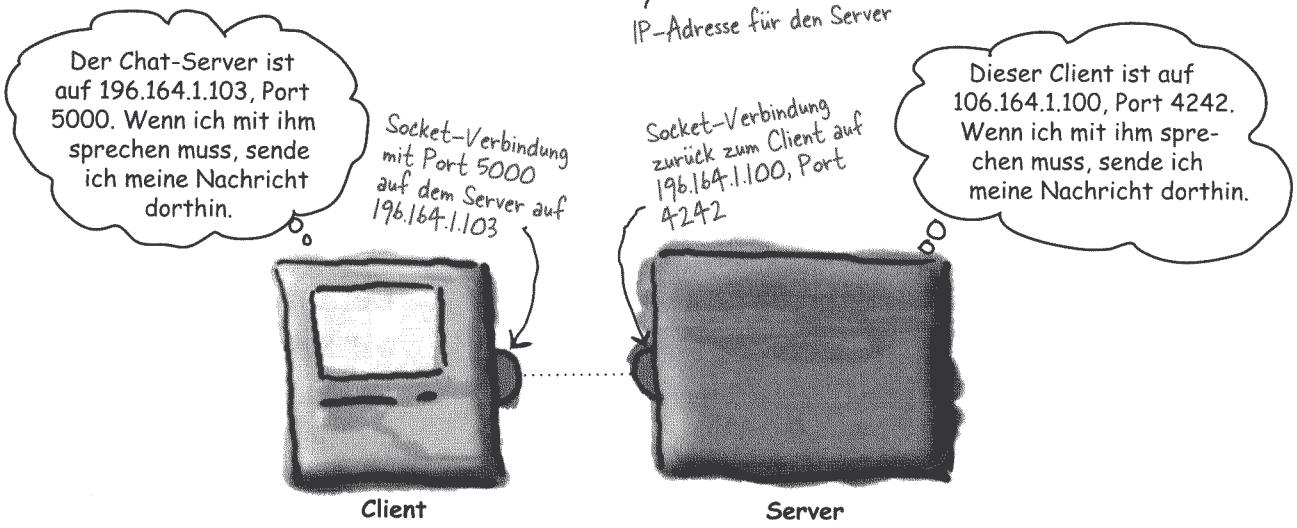
Um eine Socket-Verbindung herzustellen, müssen Sie zwei Dinge über den Server wissen: wer er ist und auf welchem Port er läuft.

Mit anderen Worten: die IP-Adresse und die TCP-Portnummer.

TCP-Portnummer

```
Socket chatSocket = new Socket("196.164.1.103", 5000);
```

IP-Adresse für den Server




Eine Socket-Verbindung bedeutet, dass die beiden Maschinen Informationen übereinander haben, einschließlich des Standorts im Netzwerk (der IP-Adresse) und des TCP-Ports.

## Ein TCP-Port ist nichts weiter als eine Zahl. Eine 16-Bit-Zahl zur Identifizierung eines bestimmten Programms auf dem Server.

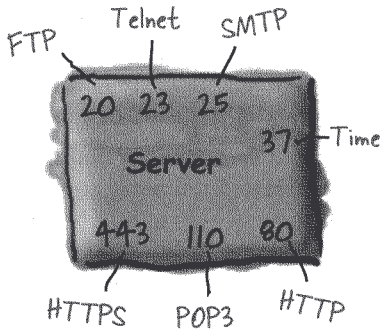
Ihr Internet-Webserver (HTTP-Server) läuft auf Port 80. Das ist ein Standard. Wenn Sie einen Telnet-Server haben, läuft er auf Port 23. FTP? 20. POP3-Mailserver? 110. SMTP? 25. Der Timeserver liegt auf 37. Betrachten Sie Portnummern als unverwechselbare Identifikationsmerkmale. Sie repräsentieren eine logische Verbindung mit einer ganz bestimmten Software auf dem Server. Das ist alles. Sie können nicht Ihren Rechner umdrehen und nach einem TCP-Port suchen. Und außerdem haben Sie auf einem Server 65536 davon (0 bis 65535). Also kann es sich dabei ganz offensichtlich nicht um einen Ort handeln, an dem reale Geräte eingestöpselt werden. Es sind einfach nur Nummern, die eine Anwendung repräsentieren.

Ohne Portnummern könnte der Server nicht wissen, mit welcher Anwendung sich ein Client verbinden möchte. Und da jede Anwendung ihr eigenes spezielles Protokoll haben könnte, überlegen Sie mal, welche Probleme Sie ohne diese Identifikationsnummern hätten. Was wäre, wenn Ihr Webbrowser beispielsweise auf dem POP3- Mailserver landen würde statt auf dem HTTP-Server? Der Mailserver wüsste nicht, wie man eine HTTP-Anfrage analysiert! Und selbst wenn er es wüsste, hätte er doch keine Ahnung, wie man eine HTTP-Anfrage bedient.

Beim Schreiben eines Server-Programms fügen Sie Code ein, der dem Programm sagt, auf welcher Portnummer es laufen soll (wie man das in Java macht, zeigen wir Ihnen etwas weiter unten in diesem Kapitel). In dem Chat-Programm, das wir in diesem Kapitel schreiben, haben wir dafür die 5000 gewählt. Absolut willkürlich. Und weil sie die Bedingung erfüllt, dass die Zahl zwischen 1024 und 65535 liegen muss. Warum 1024? Weil 0 bis 1023 für die bereits genannten »wohlbekannten« Dienste (engl. well-known services, abgekürzt WKS) reserviert sind.

Wenn Sie nun Dienste (Server-Programme) schreiben, die auf einem Firmennetz laufen, sollten Sie mit Ihren Systemadministratoren klären, welche Ports schon belegt sind. Vielleicht sagen die Admins Ihnen, dass Sie keine Portnummer unterhalb von, sagen wir, 3000 benutzen dürfen. Auf jeden Fall sollten Sie Portnummern nicht nach Lust und Laune vergeben, wenn Sie keinen Ärger riskieren wollen. Außer es geht um das Netzwerk bei Ihnen *zu Hause*. Dann müssen Sie das eben mit *Ihrem Nachwuchs* klären.

Reservierte TCP-Portnummern für verbreitete Server-Anwendungen




Auf einem Server können bis zu 65536 verschiedene Server-Anwendungen laufen, eine pro Port.

**Die TCP-Portnummern von 0 bis 1023 sind für »wohlbekannte« Dienste reserviert. Benutzen Sie sie nicht für Ihre eigenen Server-Programme!***

**Der Chat-Server, den wir schreiben, benutzt Port 5000. Wir haben einfach irgendeine Zahl zwischen 1024 und 65535 ausgewählt.**

*Na ja, Sie *könnten* schon eine davon verwenden – aber wahrscheinlich bringt der Systemadministrator Ihrer Firma Sie dann um.

## Es gibt keine Dummen Fragen

**F: Woher weiß man die Portnummer des Server-Programms, das man ansprechen will?**

A: Das hängt davon ab, ob das Programm einer der »wohlbekannten Dienste« ist. Wenn Sie sich mit einem solchen Dienst verbinden wollen, z.B. mit einem der auf der vorigen Seite genannten Dienste (HTTP, SMTP, FTP usw.), können Sie die Nummern im Internet nachschlagen (googeln Sie nach »well-known TCP Port« oder »reservierte Ports«). Oder fragen Sie den netten System-Admin in Ihrer Nachbarschaft.

Wenn das Programm jedoch nicht zu den wohlbekannten Diensten gehört, müssen Sie die Nummer bei demjenigen erfragen, der Ihren Dienst anbietet. Wenn jemand einen Netzwerkdienst schreibt und möchte, dass andere dafür Clients schreiben, veröffentlicht er normalerweise die IP-Adresse, die Portnummer und das Protokoll für den Dienst. Möchten Sie beispielsweise einen Client für einen Go-Spieleserver schreiben, können Sie auf eine der Go-Server-Seiten gehen und finden dort Informationen darüber, wie man einen Client für diesen bestimmten Server schreibt.

**F: Kann auf einem Port jemals mehr als ein Programm laufen? Anders ausgedrückt, können zwei Anwendungen auf dem gleichen Server die gleiche Portnummer haben?**

A: Nein! Wenn Sie versuchen, ein Programm an einen Port zu binden, der schon benutzt wird, erhalten Sie eine BindException. Ein Programm an einen Port zu *binden* bedeutet einfach, dass Sie eine Server-Anwendung starten und ihr sagen, sie soll auf einem bestimmten Port laufen. Mehr darüber erfahren Sie, wenn wir zum Server-Teil dieses Kapitels kommen.

IP-Adresse entspricht der Einkaufsstraße.

Die IP-Adresse ist wie die Angabe einer bestimmten Einkaufsstraße, z.B. »Rathausstraße«.

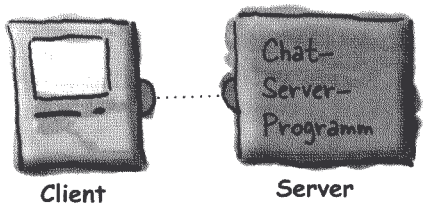
Portnummer entspricht dem bestimmten Geschäft in dieser Straße.

Die Portnummer ist wie die Angabe eines bestimmten Geschäfts, z.B. »Tims CD-Shop«.

## Gewichtheben fürs Gehirn

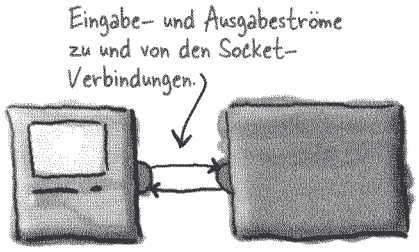
Okay, Sie haben also eine Socket-Verbindung. Der Client und der Server kennen voneinander die IP-Adresse und die TCP-Portnummer. Was jetzt? Wie kommunizieren sie über diese Verbindung? Mit anderen Worten, wie schiebt man Bits von einem zum anderen? Stellen Sie sich vor, welche Art von Nachrichten Ihr Chat-Client senden und empfangen muss.

Wie reden diese beiden eigentlich miteinander?



# Verwenden Sie einen BufferedReader, um Daten von einem Socket zu lesen

Für die Kommunikation über eine Socket-Verbindung benutzen Sie Ströme. Ganz normale E/A-Ströme, wie die aus dem letzten Kapitel. Eine der tollsten Eigenschaften von Java ist, dass es meistens nicht so darauf ankommt, womit Ihr Highlevel-Verkettungsstrom tatsächlich verbunden ist. Mit anderen Worten, Sie können einen BufferedReader genau so benutzen wie beim Schreiben in eine Datei – mit dem Unterschied, dass der darunter liegende Verbindungsstrom mit einem *Socket* verbunden ist statt mit einer *Datei!*




## 1 Eine Socket-Verbindung zum Server herstellen

```
Socket chatSocket = new Socket("127.0.0.1", 5000);
```

Die Portnummer. Sie kennen sie, weil wir Ihnen GESAGT haben, dass 5000 die Portnummer für unseren Chatserver ist.

127.0.0.1 ist die IP-Adresse für »localhost«, also den Rechner, auf dem dieser Code läuft. Diese Adresse können Sie verwenden, wenn Sie Ihren Client und Ihren Server auf einer einzigen »Stand-alone«-Maschine testen.

## 2 Einen InputStreamReader erzeugen, der mit dem Lowlevel-Eingabestrom des Sockets (dem Verbindungsstrom) verkettet ist

```
InputStreamReader strom = new InputStreamReader(chatSocket.getInputStream());
```

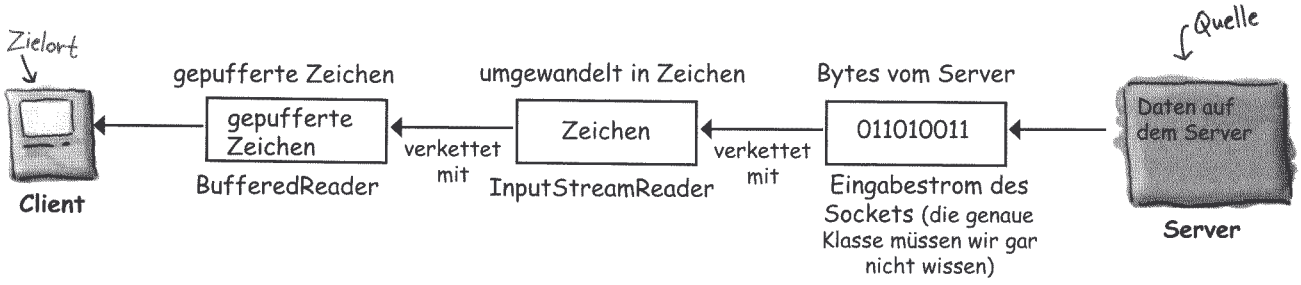
InputStreamReader ist eine »Brücke« zwischen einem Lowlevel-Bytestrom (z.B. dem vom Socket kommenden Strom) und einem Highlevel-Zeichenstrom (z.B. dem BufferedReader, den wir als obersten Strom unserer Kette einsetzen).

Wir müssen nichts weiter tun, als den Socket um einen Eingabestrom zu BITTEN! Wir bekommen einen Lowlevel-Verbindungsstrom, aber wir verketten ihn einfach mit etwas Text-freundlicherem.

## 3 Einen BufferedReader machen und lesen!

Wir verketten den BufferedReader mit dem InputStreamReader (der wiederum mit dem Lowlevel-Verbindungsstrom verkettet ist, den wir vom Socket erhalten haben).

```
BufferedReader reader = new BufferedReader(strom);
String message = reader.readLine();
```



# Verwenden Sie einen PrintWriter, um Daten in einen Socket zu schreiben

Im letzten Kapitel haben wir nicht PrintWriter verwendet, sondern BufferedWriter. Dieses Mal haben wir die Wahl, aber wenn Sie etwas String für String schreiben wollen, ist PrintWriter die Klasse der Wahl. Und die beiden wichtigsten Methoden – print() und println() – finden Sie auch beim PrintWriter wieder! Genau wie beim guten alten System.out.

## 1 Eine Socket-Verbindung zum Server erzeugen

Dieser Teil ist das Gleiche wie auf der gegenüberliegenden Seite – um an den Server zu schreiben, müssen wir uns ja mit ihm verbinden.

```
Socket chatSocket = new Socket("127.0.0.1", 5000);
```

## 2 Einen PrintWriter erzeugen, der mit dem Lowlevel-Ausgabestrom des Sockets (dem Verbindungsstrom) verkettet ist

```
PrintWriter writer = new PrintWriter(chatSocket.getOutputStream());
```

PrintWriter fungiert selbst als Brücke zwischen Daten in Zeichenform und den Bytes, die er vom Lowlevel-Ausgabestrom des Sockets erhält. Durch die Verkettung eines PrintWriter mit dem Ausgabestrom des Sockets können wir Strings in die Socket-Verbindung schreiben.

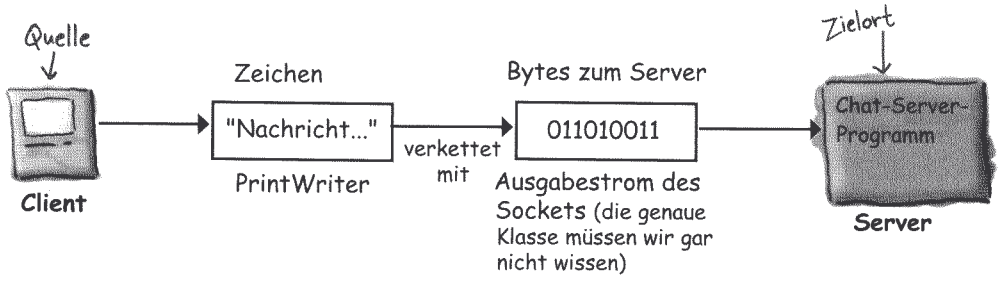
Der Socket gibt uns einen Lowlevel-Verbindungsstrom, den wir mit dem PrintWriter verketten, indem wir ihn dessen Konstruktor übergeben.

## 3 Etwas schreiben (»ausdrucken«)

```
writer.println("zu sendende Nachricht");
writer.print("noch eine Nachricht");
```

println() fügt am Ende des Gesendeten einen Zeilenumbruch ein.

print() fügt keinen Zeilenumbruch ein.



# Der TippDesTagesClient

Bevor wir uns der Programmierung der Chat-Anwendung zuwenden, wollen wir mit etwas Kleinerem beginnen. Der Ratgeberonkel ist ein Server-Programm, das praktische, anregende Tipps gibt, die Ihnen durch diese langen Code-erfüllten Tage helfen.

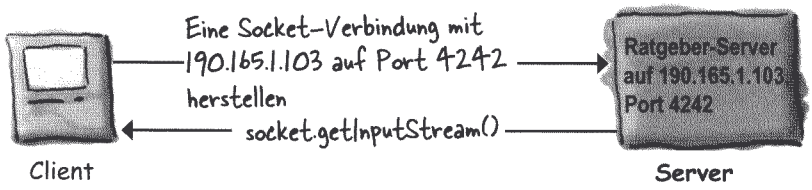
Wir bauen uns einen Client für das Ratgeberonkel-Programm, der bei jedem Verbindungsaufbau eine Nachricht vom Server holt.

Worauf warten Sie? Wer weiß, was Sie ohne dieses Programm alles schon verpasst haben!

Der Ratgeberonkel

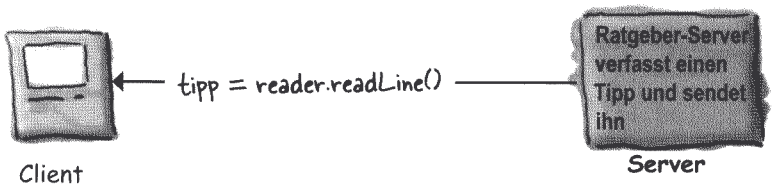
## 1 Verbindung herstellen

Der Client verbindet sich mit dem Server und erhält von ihm einen Eingabestrom.




## 2 Lesen

Der Client liest eine Nachricht vom Server.



# Der Code für den TippDesTagesClient

Dieses Programm erzeugt einen Socket und einen BufferedReader (unterstützt durch weitere Ströme) und liest eine einzelne Zeile von der Server-Anwendung (was auch immer auf Port 4242 läuft).

```
import java.io.*;
import java.net.*;

public class TippDesTagesClient {

  public void los() {
    try {
      Socket s = new Socket("127.0.0.1", 4242);

      InputStreamReader streamReader = new InputStreamReader(s.getInputStream());
      BufferedReader reader = new BufferedReader(streamReader);

      String tipp = reader.readLine();
      System.out.println("Tipp des Tages: " + tipp);

      reader.close();

    } catch(IOException ex) {
      ex.printStackTrace();
    }
  }

  public static void main(String[] args) {
    TippDesTagesClient client = new TippDesTagesClient();
    client.los();
  }
}
```

Die Klasse Socket gehört zu java.net.

Hier kann eine Menge schief gehen.

Stellen Sie eine Socket-Verbindung zu dem her, was da gerade auf Port 4242 läuft, und zwar auf dem gleichen Host, auf dem auch dieser Code hier läuft (dem >>localhost<<).

Verketten Sie einen BufferedReader mit einem InputStreamReader, der wiederum mit dem Eingabestrom vom Socket verkettet ist.

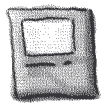
Dieses readLine() ist GENAU das Gleiche wie die Verwendung eines BufferedReader, der mit einer DATEI verkettet ist. Mit anderen Worten: Wenn Sie eine BufferedReader-Methode aufrufen, schert sich der Reader nicht darum, wo die Zeichen herkommen.

Damit werden ALLE Ströme geschlossen.

## Spitzen Sie Ihren Bleistift

Testen Sie, ob Ihnen die Ströme/Klassen für das Lesen und Schreiben von einem Socket richtig im Gedächtnis geblieben sind. Versuchen Sie, nicht auf die nächste Seite zu sehen!

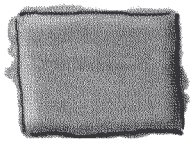
Text von einem Socket ***lesen:***



Client

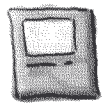
hier bitte die Kette von Strömen hinschreiben/–zeichnen, die der Client zum Lesen vom Server benutzt

Quelle



Server

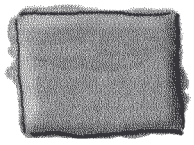
Text an einen Socket ***senden:***



Client

hier bitte die Kette von Strömen hinschreiben/–zeichnen, mit denen der Client an den Server sendet

Zielort



Server

## Spitzen Sie Ihren Bleistift

**Füllen Sie die Lücken aus:**

Welche beiden Informationen braucht der Client, um eine Socket-Verbindung zu einem Server herzustellen? _____ _____

Welche TCP-Portnummern sind für »wohlbekannte Dienste« wie HTTP und FTP reserviert? _____

WAHR oder FALSCH: Der Bereich der gültigen TCP-Portnummern kann durch einen primitiven short-Wert repräsentiert werden. _____

# Einen einfachen Server schreiben

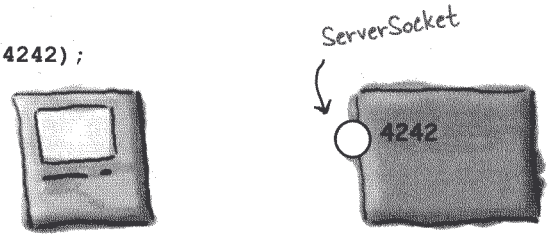
Was braucht man also, um eine Server-Anwendung zu schreiben? Nichts weiter als ein Paar Sockets. Ja, ein *Paar* mit großem »P« – also zwei. Einen ServerSocket, der auf Client-Anfragen wartet (wenn ein Client einen »new Socket()« erzeugt), und einen einfachen Socket, der für die Kommunikation mit dem Client benutzt wird.

## Und so funktioniert es:

**1** Die Server-Anwendung erzeugt einen ServerSocket auf einem ganz bestimmten Port.

```
ServerSocket serverSock = new ServerSocket(4242);
```

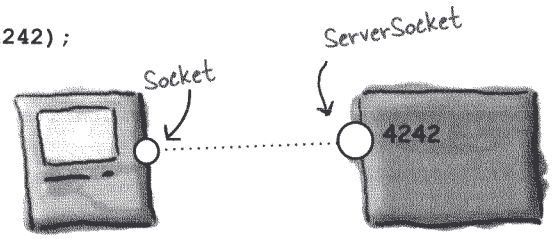
Damit beginnt die Server-Anwendung, auf eingehende Client-Anfragen auf Port 4242 zu lauschen.




**2** Der Client stellt eine Socket-Verbindung zur Server-Anwendung her.

```
Socket sock = new Socket("190.165.1.103", 4242);
```

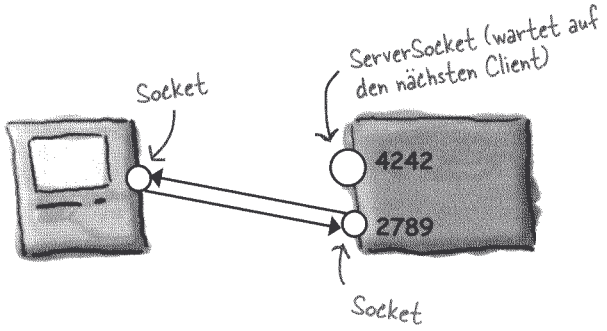
Der Client kennt die IP-Adresse und die Portnummer (veröffentlicht oder von demjenigen erfragt, der die Server-Anwendung so konfiguriert hat, dass sie genau auf diesem Port liegt).




**3** Der Server erzeugt einen neuen Socket für die Kommunikation mit diesem Client.

```
Socket sock = serverSock.accept();
```

Die accept()-Methode blockiert (sie sitzt einfach da), während sie auf eine Client-Socket-Verbindung wartet. Wenn dann schließlich ein Client versucht, sich zu verbinden, gibt die Methode einen einfachen Socket zurück (auf einem *anderen* Port), der weiß, wie er mit dem Client kommuniziert (d.h., er kennt die IP-Adresse und Portnummer des *Clients*). Dieser Socket liegt auf einem anderen Port als der ServerSocket, so dass dieser sich wieder dem Warten auf andere Clients zuwenden kann.



# TippDesTagesServer-Code

Dieses Programm macht einen ServerSocket und wartet auf Client-Anfragen. Wenn es eine Client-Anfrage erhält (der Client also new Socket() sagt), stellt der Server eine neue Socket-Verbindung zu diesem Client her. Der Server erzeugt einen PrintWriter (mit dem Ausgabestrom des Sockets) und sendet eine Nachricht an den Client.

```
import java.io.*;
import java.net.*;

public class TippDesTagesServer {

  String[] tippListe = {"Essen Sie weniger Chips und Gummibärchen.", "Holen Sie sich
die engen Jeans. Nein, Sie sehen darin NICHT dick aus.", "Mit einem Wort: unmöglich!",
"Seien Sie ehrlich, nur heute. Sagen Sie Ihrem Chef, was Sie *wirklich* denken.", "Sie
sollten wirklich mal wieder zum Friseur gehen ..."};

  public void los() {

    try {

      ServerSocket serverSock = new ServerSocket(4242);

      while(true) {

        Socket sock = serverSock.accept();

        PrintWriter writer = new PrintWriter(sock.getOutputStream());
        String tipp = getTipp();
        writer.println(tipp);
        writer.close();
        System.out.println(tipp);
      }

    } catch(IOException ex) {
      ex.printStackTrace();
    }
  } // los() schließen

  private String getTipp() {
    int zufallszahl = (int) (Math.random() * tippListe.length);
    return tippListe[zufallszahl];
  }

  public static void main(String[] args) {
    TippDesTagesServer server = new TippDesTagesServer();
    server.los();
  }
}
```

Denken Sie an die Importe.

Der TippDesTages kommt aus diesem Array.

(Denken Sie daran, die Zeilenumbrüche innerhalb dieser Strings macht der Code-Editor. Niemals mitten in einem String die Return-Taste drücken!)

Der ServerSocket sorgt dafür, dass diese Server-Anwendung auf dem Rechner, auf dem dieser Code läuft, auf Port 4242 auf Client-Anfragen »lauscht«.

Der Server geht in eine permanente Schleife, in der er auf Client-Anfragen wartet (und sie bedient).

Die accept-Methode blockiert (sie sitzt einfach da), bis eine Anfrage hereinkommt. Dann gibt die Methode einen Socket für die Kommunikation mit dem Client zurück (auf irgendeinem anonymen Port).

Jetzt erzeugen wir mit der Socket-Verbindung zum Client einen PrintWriter und senden an diesen (über println()) eine String-Ratgeberbotschaft. Dann schließen wir den Socket wieder, weil wir mit diesem Client fertig sind.

# Gewichtheben fürs Gehirn

***Woher weiß der Server, wie er mit dem Client kommuniziert?***

Der Client kennt die IP-Adresse und die Portnummer des Servers, aber wie kann der Server eine Socket-Verbindung mit dem Client herstellen (und Eingabe- sowie Ausgabeströme erzeugen)?

Denken Sie daran, wann, wo und wie der Server etwas über den Client erfährt.

## Es gibt keine Dummen Fragen

**F: Der Ratgeber-Servercode auf der gegenüberliegenden Seite hat einen gewaltigen Nachteil – es sieht so aus, als könnte er nur einen einzigen Client auf einmal bedienen!**

A: Ja, das stimmt. Er kann keine Anfrage von einem neuen Client entgegennehmen, bevor er nicht mit dem aktuellen Client fertig und in den nächsten Durchlauf der unendlichen Schleife eingetreten ist. (Wo er wieder so lange im accept()-Aufruf sitzen bleibt, bis eine Anfrage hereinkommt; dann erzeugt er einen Socket für den neuen Client, und alles geht von vorne los.)

**F: Ich will die Frage noch mal anders formulieren: Wie kann man einen Server machen, der mehrere Clients auf einmal bedienen kann??? Das hier würde doch zum Beispiel bei einem Chat-Server *nie* funktionieren.**

A: Aaah, das ist wirklich einfach. Benutzen Sie mehrere Threads und übergeben Sie jeden neuen Client-Socket an einen anderen Thread. Wie man das macht, lernen wir jetzt gleich!

## Punkt für Punkt

- Client- und Server-Anwendungen kommunizieren über eine Socket-Verbindung.
- Ein Socket repräsentiert eine Verbindung zwischen zwei Anwendungen, die auf zwei verschiedenen realen Maschinen laufen können (aber nicht müssen).
- Ein Client muss die IP-Adresse (oder den Domain-Namen) und die TCP-Portnummer der Server-Anwendung wissen.
- Ein TCP-Port ist eine vorzeichenlose 16-Bit-Zahl, die einer bestimmten Server-Anwendung zugeordnet ist. Mit Hilfe von TCP-Portnummern können sich verschiedene Clients mit derselben Maschine verbinden und dabei mit unterschiedlichen auf diesem Rechner laufenden Anwendungen kommunizieren.
- Die Portnummern von 0 bis 1023 sind für »wohlbekannte Dienste« (well-known services) reserviert, unter anderem HTTP, FTP, SMTP usw.
- Ein Client verbindet sich mit einem Server, indem er einen Socket erzeugt:

```
Socket s = new Socket("127.0.0.1", 4200);
```

- Ist die Verbindung einmal hergestellt, kann ein Client vom Socket Eingabe- und Ausgabeströme bekommen. Dabei handelt es sich um Lowlevel-»Verbindungsströme«.

```
s.getInputStream();
```

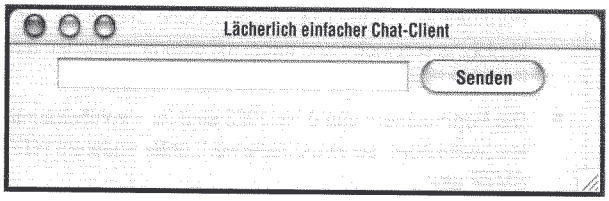
- Um Textdaten vom Server zu lesen, erzeugen Sie einen BufferedReader, der mit einem InputStreamReader verkettet ist, der wiederum mit dem Eingabestrom vom Socket verbunden ist.
- InputStreamReader ist ein »Brückenstrom«, der Bytes entgegennimmt und sie in (aus Zeichen bestehende) Textdaten umwandelt. Er dient in erster Linie als mittleres Kettenglied zwischen dem Highlevel-BufferedReader und dem Lowlevel-Socket-Eingabestrom.
- Um Textdaten an den Server zu schreiben, erzeugen Sie einen PrintWriter, der direkt mit dem Ausgabestrom des Sockets verkettet ist. Rufen Sie die Methode print() oder println() auf, um Strings an den Server zu senden.
- Server verwenden einen ServerSocket, der auf einer bestimmten Portnummer auf Client-Anfragen wartet.
- Wenn bei einem ServerSocket eine Anfrage eingeht, »akzeptiert« dieser die Anfrage, indem er eine Socket-Verbindung zu dem Client herstellt.

# Einen Chat-Client schreiben

Wir schreiben die Chat-Client-Anwendung in zwei Stufen. Als Erstes machen wir eine Nur-senden-Version, die Nachrichten an den Server sendet, aber keine der Nachrichten von den anderen Teilnehmern zu Gesicht bekommt (welch spannende und geheimnisvolle Wendung im Chatroom-Konzept!).

Anschließend steigen wir dann aber in das volle Programm ein und machen einen Chatroom, der Nachrichten sowohl senden als auch empfangen kann.

## Version 1: Nur senden




Tippen Sie eine Nachricht ein, drücken Sie dann auf >>Senden<< und schicken Sie sie damit an den Server. In dieser Version erhalten wir keine Nachrichten VOM Server, deshalb gibt es noch keinen scrollbaren Textbereich.

## Code grob skizziert

```
public class EinfacherChatClientA {

  JTextField ausgehend;
  PrintWriter writer;
  Socket sock;

  public void los() {
    // GUI machen und einen Listener beim Senden-Button registrieren
    // die Methode netzwerkEinrichten() aufrufen
  }

  private void netzwerkEinrichten() {
    // einen Socket und dann einen PrintWriter machen
    // den PrintWriter der Instanzvariablen writer zuweisen
  }

  private class SendenButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // den Text aus dem Textfeld holen und
      // ihn mit dem writer (einem PrintWriter) an den Server senden
    }
  } // innere Klasse SendenButtonListener schließen

} // äußere Klasse schließen
```

```
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class EinfacherChatClientA {

  JTextField ausgehend;
  PrintWriter writer;
  Socket sock;

  public static void main(String[] args) {
    EinfacherChatClientA client = new EinfacherChatClientA();
    client.los();
  }

  public void los() {
    JFrame frame = new JFrame("Lächerlich einfacher Chat-Client");
    JPanel hauptPanel = new JPanel();
    ausgehend = new JTextField(20);
    JButton sendenButton = new JButton("Senden");
    sendenButton.addActionListener(new SendenButtonListener());
    hauptPanel.add(ausgehend);
    hauptPanel.add(sendenButton);
    netzwerkEinrichten();
    frame.getContentPane().add(BorderLayout.CENTER, hauptPanel);
    frame.setSize(400,500);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  } // los schließen

  private void netzwerkEinrichten() {
    try {
      sock = new Socket("127.0.0.1", 5000);
      writer = new PrintWriter(sock.getOutputStream());
      System.out.println("Netzwerkverbindung steht");
    } catch(IOException ex) {
      ex.printStackTrace();
    }
  } // netzwerkEinrichten schließen

  private class SendenButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      try {
        writer.println(ausgehend.getText());
        writer.flush();

      } catch(Exception ex) {
        ex.printStackTrace();
      }
      ausgehend.setText("");
      ausgehend.requestFocus();
    }
  } // innere Klasse SendenButtonListener schließen
}
```

Importe für die Ströme (java.io), den Socket (java.net) und den GUI-Kram.

Hier wird die GUI aufgebaut, nichts Neues für uns und nichts, was mit Netzwerk oder E/A zusammenhängt.

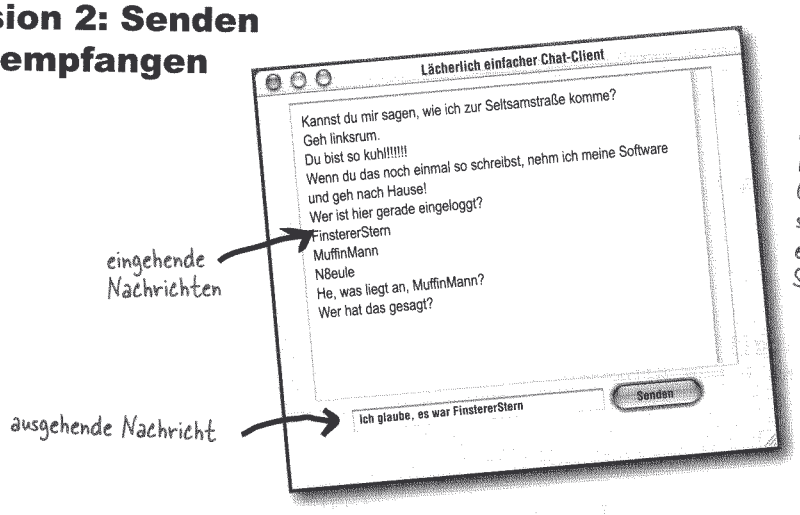
Wir benutzen den Localhost, damit Sie den Client und den Server auf einund demselben Rechner testen können.

Hier machen wir den Socket und den PrintWriter (das wird aus der Methode los() heraus aufgerufen, direkt bevor die Benutzeroberfläche angezeigt wird).

Jetzt erledigen wir das eigentliche Schreiben. Erinnern Sie sich, der writer ist mit dem Eingabestrom vom Socket verkettet; bei jedem println() geht es also über das Netz zum Server!

**Falls Sie dies jetzt gleich ausprobieren möchten, tippen Sie den fertig vorbereiteten Chat-Server-Code ab, der am Ende dieses Kapitels abgedruckt ist. Als Erstes starten Sie den Server. Öffnen Sie anschließend ein weiteres Terminal und starten Sie darin diesen Client.**

## Version 2: Senden und empfangen




Der Server sendet eine Nachricht an alle Client-Teilnehmer, sobald er die Nachricht empfangen hat. Wenn ein Client eine Nachricht sendet, taucht sie erst dann im Display-Bereich für eingehende Nachrichten auf, wenn der Server sie an alle verschickt.

### Die große Frage: *WIE* erhalten Sie Nachrichten vom Server?

Das sollte nicht schwer sein – wenn Sie das Netzwerk einrichten, erzeugen Sie auch einen Eingabestrom (wahrscheinlich einen BufferedReader). Dann lesen Sie Nachrichten mit readLine().

### Eine noch größere Frage: *WANN* erhalten Sie Nachrichten vom Server?

Überlegen Sie mal. Welche Möglichkeiten gibt es?

#### ① Möglichkeit eins: Den Server alle 20 Sekunden abfragen

**Pro:** Okay, es ist machbar.

**Kontra:** Woher soll der Server wissen, was Sie schon gesehen haben und was nicht? Der Server müsste die Nachrichten speichern, statt sie jedes Mal, wenn eine eingeht, einfach zu verteilen und anschließend zu vergessen. Und warum 20 Sekunden? Eine solche Verzögerung beeinträchtigt die Benutzerfreundlichkeit, aber wenn Sie die Verzögerung kleiner machen, riskieren Sie unnötige Zugriffe auf Ihren Server. Ineffizient.

#### ② Möglichkeit zwei: Jedes Mal, wenn der Benutzer eine Nachricht verschickt, vom Server lesen

**Pro:** Machbar, sehr einfach.

**Kontra:** Es ist dumm. Warum ein solch willkürlich gewähltes Zeitintervall für die Prüfung auf Nachrichten? Was ist, wenn der Benutzer nur passiv mitliest und selbst nie etwas sendet?

#### ③ Möglichkeit drei: Nachrichten lesen, sobald sie vom Server gesendet werden

**Pro:** Sehr effizient und benutzerfreundlich.

**Kontra:** Wie macht man denn zwei Dinge gleichzeitig? Wo würden Sie diesen Code hinschreiben? Sie würden irgendwo eine Schleife brauchen, die ständig darauf wartet, vom Server zu lesen. Aber wo sollte das sein? Wenn Sie die Oberfläche einmal gestartet haben, passiert ja erst wieder etwas, wenn eine GUI-Komponente ein Event absetzt.

In Java können Sie tatsächlich zur gleichen Zeit laufen und Kaugummi kauen.

## Inzwischen dürfte Ihnen klar sein, dass wir uns für Möglichkeit drei entscheiden.

Wir möchten etwas, das kontinuierlich läuft und auf Nachrichten vom Server prüft, aber *ohne dass die Fähigkeit des Benutzers zur Interaktion mit dem GUI beeinträchtigt wird!* Während also der Benutzer fröhlich neue Nachrichten eintippt oder durch die erhaltenen Nachrichten scrollt, möchten wir *hinter den Kulissen* etwas haben, das vom Server eingehende Nachrichten liest.

Das heißt, wir brauchen letztendlich eine Art »Nebenhandlung«: einen neuen Thread. Einen neuen, separaten Stack.

Alles, was wir in der Nur-senden-Version (Version 1) gemacht haben, soll noch wie vorher funktionieren, während nebenher ein neuer *Prozess* ausgeführt wird, der Informationen vom Server liest und sie in dem Textbereich für eingehende Nachrichten anzeigt.

Na ja, nicht so ganz. Falls Sie nicht gerade mehrere Prozessoren in Ihrem Rechner haben, ist ein neuer Java-Thread in Wirklichkeit kein separater Prozess auf dem Betriebssystem. Aber man könnte beinahe *glauben*, er wäre es.

## Multithreading in Java

Bei Java ist das Multithreading, also die Ausführung mehrerer paralleler »Handlungsstränge«, schon fest in die Sprache eingebaut. Und ein neuer Ausführungsstrang ist fix gemacht:

```
Thread t = new Thread();
t.start();
```

Das war's. Durch die Erzeugung eines neuen Thread-*Objekts* haben Sie einen separaten *Ausführungsstrang – einen Thread* – »vom Stapel gelassen«, mit einem eigenen Aufruf-Stack.

***Da wäre aber noch ein Problem.***

Dieser Thread tut rein gar nichts und »stirbt« daher im Moment seiner Geburt auch schon seinen »virtuellen Tod«. Wenn ein Thread stirbt, verschwindet auch sein neuer Stack wieder. Ende der Geschichte.

Es fehlt also ein wichtiger Bestandteil – die *Aufgabe* des Threads. Mit anderen Worten, wir brauchen auch den Code, den ein separater Thread ausführen soll.

In Java bedeutet Multithreading, dass wir uns sowohl den *Thread* als auch die *Aufgabe* ansehen müssen, den er *ausführt*. Und wir müssen uns die *Klasse* Thread im Paket java.lang ebenfalls ansehen. (Erinnern Sie sich, java.lang ist das Paket, das immer kostenlos und unaufgefordert importiert wird; dort tummeln sich die Klassen, die für die Sprache am grundlegendsten sind, einschließlich String und System.)

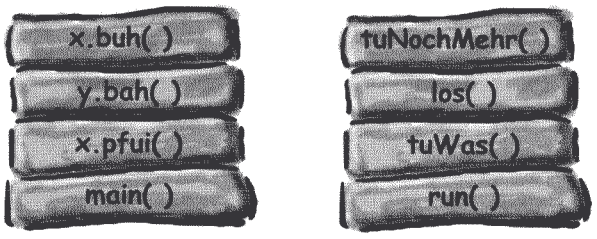
# Verwechseln Sie Threads im allgemeinen Sinn nicht mit der Klasse Thread

Wir können über *Threads* im allgemeinen Sinn sprechen und über die Klasse **Thread**. Wenn allgemein von einem *Thread* die Rede ist, ist ein separater *Ausführungsstrang* gemeint, also ein »Handlungsfaden« mit einem eigenen Geschehen. Mit anderen Worten: ein separater Aufruf-Stack. Es gibt aber auch eine Klasse namens **Thread** im Paket java.lang. Ein **Thread**-Objekt repräsentiert einen Ausführungs-Thread. Jedes Mal, wenn Sie einen neuen *Ausführungs-Thread* starten wollen, werden Sie dafür eine Instanz der Klasse **Thread** erzeugen.

Ein Thread ist ein separater »Ausführungsstrang«, also ein separater Aufruf-Stack.

Ein Thread wird in Java von einem Objekt der Klasse Thread repräsentiert.

### ein Thread im allgemeinen Sinn

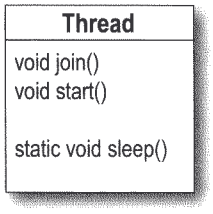



**main-Thread**

**ein anderer Thread, der vom Code gestartet wurde**

Ein Thread ist ein separater Ausführungsstrang. Ein separater Aufruf-Stack also. Jede Java-Anwendung startet einen main-Thread – den Thread, der main() als unterste Methode auf den Stack setzt. Die JVM ist dafür verantwortlich, den main-Thread zu starten (und weitere Threads, wenn es ihr gefällt, einschließlich des Garbage Collection-Threads). Als Programmierer können Sie Code schreiben, um zusätzlich eigene Threads zu starten.

### die Klasse Thread




**java.lang.Thread**

Thread ist eine Klasse, deren Instanzen einen Ausführungsstrang repräsentieren. Sie enthält Methoden, um einen Thread zu starten, um einen Thread mit einem anderen zu verbinden und um einen Thread »schlafen zu legen«. (Sie enthält noch weitere Methoden; dies hier sind nur die wichtigsten, die wir jetzt brauchen.)

# Was heißt es, mehr als einen Aufruf-Stack zu haben?

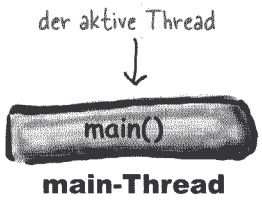
Mit mehr als einem Aufruf-Stack haben Sie den *Eindruck*, dass mehrere Dinge zur gleichen Zeit geschehen. In Wirklichkeit kann nur ein echtes Multiprozessorsystem tatsächlich mehr als eine Sache gleichzeitig tun, aber mit Java-Threads kann es so aussehen, als täte man verschiedene Dinge gleichzeitig. Mit anderen Worten, die Ausführung kann zwischen den Stacks so schnell hin- und herwechseln, dass der Eindruck erweckt wird, alle Stacks würden zur gleichen Zeit ausgeführt. Erinnern Sie sich, Java ist nur ein Prozess, der auf Ihrem darunter liegenden Betriebssystem läuft. Erst einmal muss also Java *selbst* der »aktuell ausführende Prozess« auf dem Betriebssystem sein. Aber wenn Java einmal mit der Ausführung an der Reihe ist, *was* genau führt die JVM dann aus? Welcher Bytecode wird ausgeführt? Die Antwort lautet: das, was gerade oben auf dem aktuell ausgeführten Stack liegt! Und 100 Millisekunden später könnte der aktuell ausgeführte Code zu einer *anderen* Methode auf einem *anderen* Stack hinüberwechseln.

Zu den Aufgaben eines Threads gehört es unter anderem zu verfolgen, welche Anweisung (in welcher Methode) gegenwärtig auf seinem Stack ausgeführt wird.

Das könnte ungefähr so aussehen:

**1** **Die JVM ruft main() auf.**

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

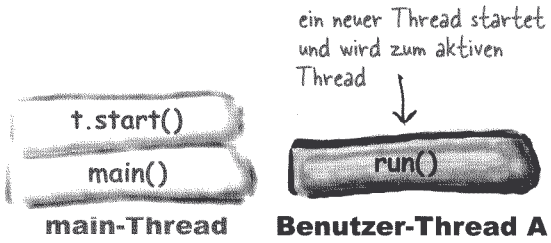



---

**2** **main() startet einen neuen Thread. Der main-Thread wird vorübergehend eingefroren, während der neue Thread zu laufen beginnt.**

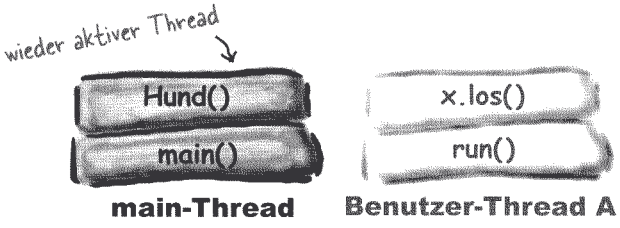
```
Runnable r = new MeinThreadJob();
Thread t = new Thread(r);
t.start();
Hund h = new Hund();
```

was das bedeutet, werden Sie gleich sehen ...




---

**3** **Die JVM wechselt zwischen dem neuen Thread (Benutzer-Thread A) und dem ursprünglichen main-Thread hin und her, bis beide Threads fertig sind.**

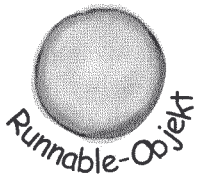


# Wie man einen neuen Thread zur Ausführung bringt:

## 1 Ein Runnable-Objekt machen (den Job des Threads)

```
Runnable threadJob = new MeinRunnable();
```

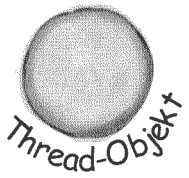
Runnable ist ein Interface, über das Sie auf der nächsten Seite etwas erfahren werden. Sie schreiben eine Klasse, die das Interface Runnable implementiert, und in dieser Klasse legen Sie fest, welche Arbeit der Thread erledigen wird. Mit anderen Worten: welche Methode von dem neuen Aufruf-Stack des Threads aus ausgeführt wird.




## 2 Ein Thread-Objekt (den Arbeiter) machen und ihm ein Runnable (den Job) geben

```
Thread meinThread = new Thread(threadJob);
```

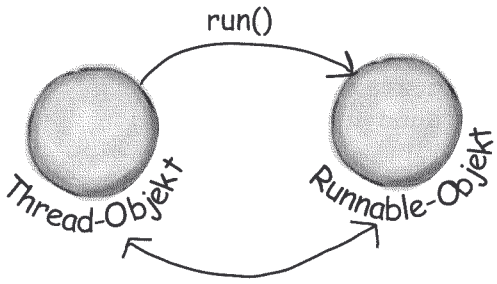
Übergeben Sie dem Thread-Konstruktor das neue Runnable-Objekt. So erfährt das neue Thread-Objekt, welche Methode er als unterste auf seinen neuen Stack setzen soll - die run()-Methode des Runnable-Objekts.




## 3 Den Thread starten

```
meinThread.start();
```

Es geschieht erst dann etwas, wenn Sie die Methode start() des Threads aufrufen. Das ist der Punkt, an dem Sie von einer bloßen Thread-Instanz zu einem neuen Ausführungs-Thread übergehen. Wenn der neue Thread startet, nimmt er die run()-Methode des Runnable-Objekts und setzt sie zuunterst auf seinen Stack.



# Jeder Thread braucht einen Job. Eine Methode, die auf seinen neuen Stack gesetzt wird.

Runnable ist für einen Thread das, was ein Job für eine Arbeitskraft ist. Ein Runnable ist die Aufgabe, die ein Thread erledigen soll.

Ein Runnable enthält die Methode, die zuunterst auf den Stack des neuen Threads kommt: run().

Ein Thread-Objekt braucht einen Job. Einen Job, den der Thread ausführt, wenn er gestartet wird. Dieser Job ist die erste Methode, die auf den Stack des neuen Threads kommt, und sie muss immer so aussehen:

```
public void run() {
  // Code, der vom neuen Thread ausgeführt wird
}
```

Das Interface Runnable definiert nur eine einzige Methode, public void run(). Erinnern Sie sich: Da das ein Interface ist, ist die Methode öffentlich, selbst wenn Sie das nicht extra dazuschreiben.

Woher der Thread weiß, welche Methode er ganz unten auf seinen Stack setzt? Er weiß es, weil das in Runnable vertraglich festgelegt ist. Weil Runnable ein Interface ist. Der Job eines Threads kann in jeder Klasse definiert werden, die das Interface Runnable implementiert. Für den Thread ist nur wichtig, dass Sie seinem Konstruktor ein Objekt einer Klasse übergeben, die Runnable implementiert.

Wenn Sie einem Thread-Konstruktor ein Runnable übergeben, geben Sie dem Thread damit einfach eine Möglichkeit, an eine run()-Methode zu gelangen. Sie geben dem Thread seinen Job.

# Um einen Job für Ihren Thread zu erzeugen, implementieren Sie das Interface Runnable

Runnable gehört zum Paket java.lang; sie müssen dieses also nicht importieren.

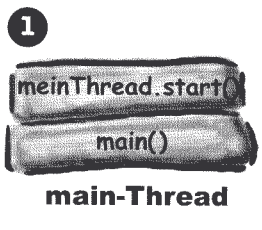
```
public class MeinRunnable implements Runnable {

  public void run() {
    los();
  }

  public void los() {
    tuNochMehr();
  }

  public void tuNochMehr() {
    System.out.println("oben auf dem Stack");
  }
}
```

Runnable enthält nur eine einzige zu implementierende Methode: public void run() (ohne Argumente). Hier schreiben Sie den JOB hinein, den der Thread erledigen soll. Dies ist die Methode, die ganz unten auf den neuen Stack kommt.

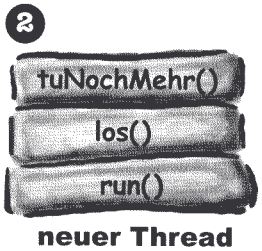
```
class ThreadTestlauf {

  public static void main (String[] args) {

    Runnable threadJob = new MeinRunnable();
    Thread meinThread = new Thread(threadJob);

    meinThread.start();

    System.out.println("zurück in main");
  }
}
```

Übergeben Sie dem neuen Thread-Konstruktor die neue Runnable-Instanz. Damit sagen Sie dem Thread, welche Methode er unten auf den neuen Stack setzen soll. Mit anderen Worten: welche Methode der neue Thread als Erstes ausführen soll.

Sie erhalten erst dann einen neuen Ausführungsstrang, wenn Sie start() auf der Thread-Instanz aufrufen. Ein Thread ist kein richtiger Thread, bevor Sie ihn gestartet haben. Davor ist er einfach eine Instanz der Klasse Thread, wie jedes andere Objekt, aber ihm fehlt die richtige >>Threadhaftigkeit<<.



main-Thread

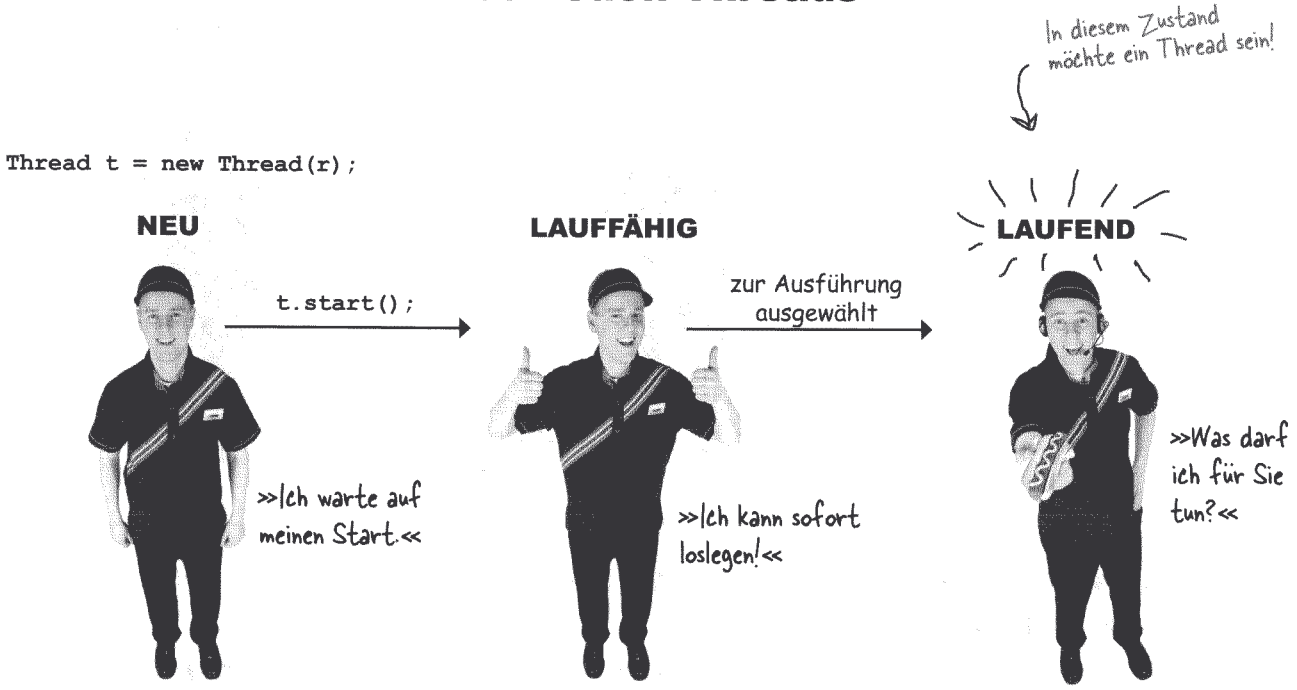


neuer Thread

## Gewichtheben fürs Gehirn

Wie sieht Ihrer Meinung nach die Ausgabe aus, wenn Sie die ThreadTestlauf-Klasse ausführen? (In ein paar Seiten werden wir das herausfinden.)

# Die drei Zustände eines neuen Threads




**`Thread t = new Thread(r);`**

Eine **neue** Thread-Instanz ist erzeugt, aber noch nicht gestartet worden. Wir haben damit zwar ein Thread-*Objekt*, aber keinen *in Ausführung befindlichen Thread.*

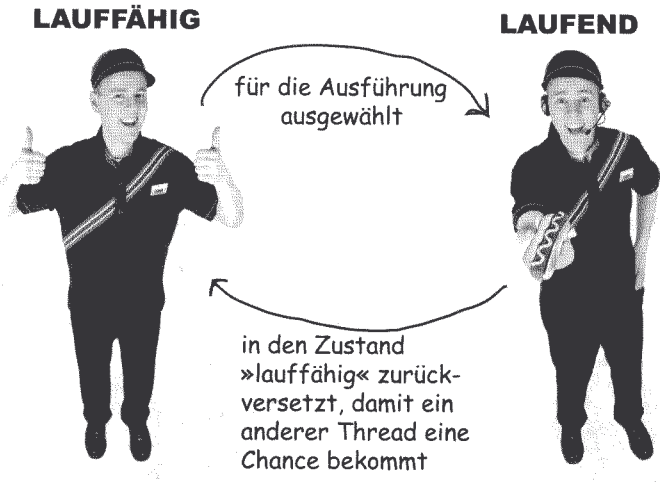
**`t.start();`**

Wenn Sie den Thread starten, wird er **lauffähig (runnable)**. In diesem Zustand ist er bereit für die Ausführung und wartet nur auf seine große Chance – darauf, dass er zur Ausführung ausgewählt wird. Ab jetzt ist für diesen Thread ein neuer Aufruf-Stack vorhanden.

Dies ist der Zustand, nach dem alle Threads lechzen: der *Auserwählte* zu sein! Der jetzt – in diesem Moment – **laufende (running)** Thread. Die Entscheidung liegt ganz allein beim *Thread-Scheduler* der JVM, der für die Ausführungsplanung verantwortlich ist. Sie können seine Entscheidung manchmal *beeinflussen,* aber Sie können den Wechsel eines Threads von »lauffähig« zu »laufend« nicht erzwingen. Im Zustand »laufend« besitzt ein Thread (und NUR dieser Thread) einen aktiven Aufruf-Stack, und die Methode oben auf dem Stack wird ausgeführt.

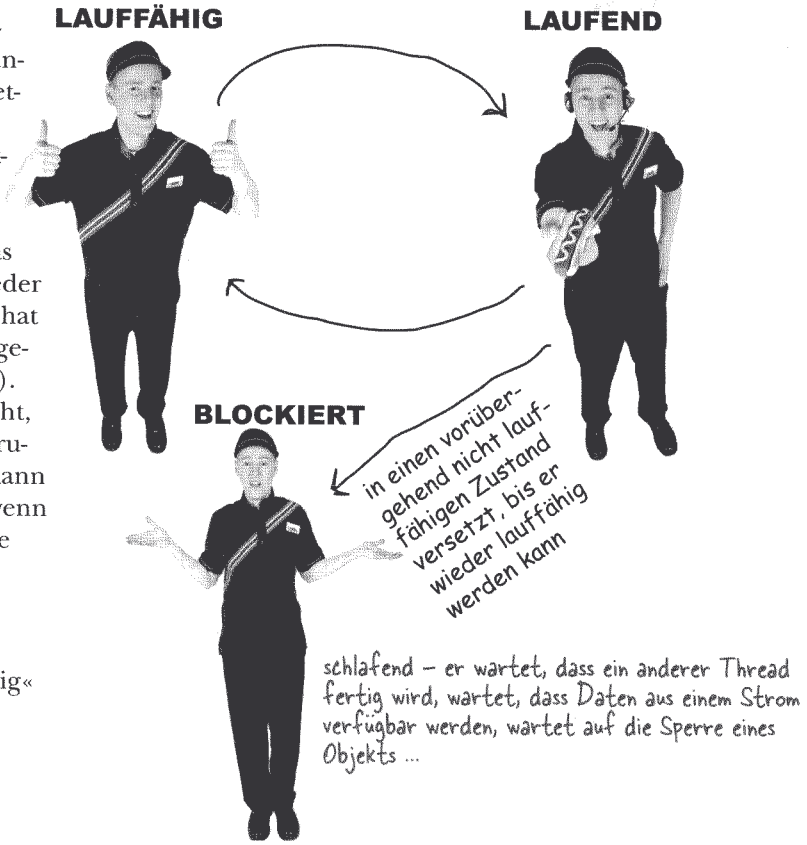
**Aber das ist noch nicht alles. Ist der Thread erst einmal lauffähig, kann er zwischen »lauffähig«, »laufend« und einem zusätzlichen Zustand »vorübergehend nicht lauffähig« (auch als »blockiert« bezeichnet) hin und her wechseln.**

## Eine typische lauffähig/laufend-Schleife

Im typischen Fall wechselt ein Thread zwischen lauffähig und laufend hin und her, weil der Thread-Scheduler der JVM einen Thread für die Ausführung auswählt, ihn dann aber wieder zurück »auf die Bank« setzt, um einem anderen Thread eine Chance zu geben.

## Ein Thread kann »vorübergehend nicht lauffähig« gemacht werden

Der Thread-Scheduler kann einen laufenden Thread aus verschiedenen Gründen in einen **blockierten Zustand** versetzen. Der Thread könnte beispielsweise Code ausführen, der aus einem Socket-Eingabestrom liest – aber es sind gerade keine Daten zum Lesen da. Der Scheduler nimmt dann den Thread aus dem Zustand »laufend« heraus, bis wieder Lesestoff verfügbar ist. Oder vielleicht hat der ausführende Code den Thread angewiesen, sich schlafen zu legen (sleep()). Oder der Thread wartet, weil er versucht, eine Methode auf einem Objekt aufzurufen, das »gesperrt« ist. In diesem Fall kann der Thread erst dann weitermachen, wenn die »Sperre« durch den Thread, der sie gerade benutzt, freigegeben wird.

Unter den genannten (und weiteren) Bedingungen wird ein Thread in den Zustand »vorübergehend nicht lauffähig« versetzt.



# Der Thread-Scheduler

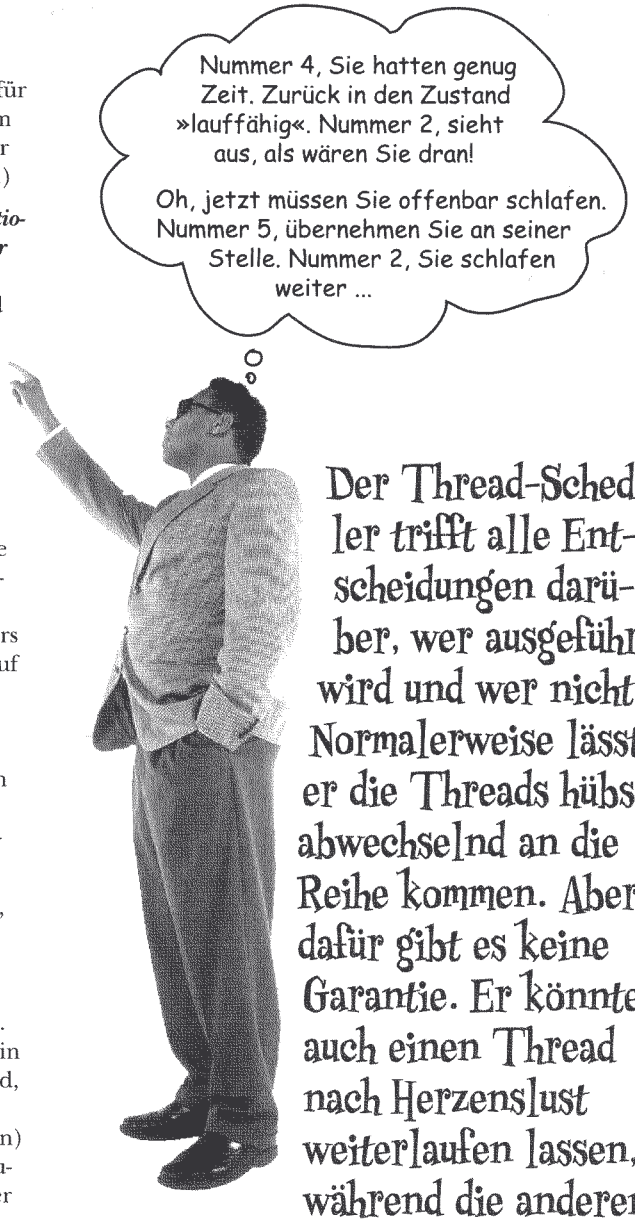
Der Thread-Scheduler ist für die Planung verantwortlich und trifft alle Entscheidungen darüber, welcher Thread vom Zustand »lauffähig« in den Zustand »laufend« versetzt wird und wann (und unter welchen Umständen) ein Thread nicht mehr »laufend« ist. Der Scheduler entscheidet, wer wie lange läuft und was aus den Threads wird, wenn er beschließt, sie aus dem Zustand »laufend« herauszunehmen.

Den Scheduler können Sie nicht steuern. Es gibt keine API für den Aufruf von Methoden auf dem Scheduler. Und vor allem gibt es bei der Planung keinerlei Garantien. (Es gibt ein paar *Fast*-Garantien, aber selbst die sind ein bisschen schwammig.)

Die Schlussfolgerung daraus ist: ***Machen Sie das richtige Funktionieren Ihres Programms nicht davon abhängig, dass der Scheduler in einer bestimmten Art und Weise arbeitet!*** Die Scheduler sind auf verschiedenen JVMs unterschiedlich implementiert, und selbst bei der Ausführung des gleichen Programms auf ein und derselben Maschine können Sie verschiedene Ergebnisse erhalten. Zu den größten Fehlern von Java-Programmieranfängern gehört es, dass sie ihre Multithreading-Programme auf nur einer einzigen Maschine testen und dann annehmen, dass der Thread-Scheduler immer genau auf diese Weise arbeiten wird, egal wo das Programm läuft.

Was bedeutet das also im Hinblick auf die Philosophie »write once, run anywhere«? Es bedeutet: Wenn Sie plattformunabhängigen Code schreiben wollen, muss Ihr Multithreading-Programm unabhängig vom Verhalten des Thread-Schedulers funktionieren. Sie dürfen sich also beispielsweise nicht darauf verlassen, dass der Scheduler dafür sorgt, dass alle Threads schön fair und gleichberechtigt nacheinander an die Reihe kommen. Auch wenn das heutzutage höchst unwahrscheinlich ist, könnte Ihr Programm doch auf einer JVM mit einem Scheduler landen, der sagt: »Okay, Thread 5, Sie sind dran, und meinetwegen können Sie weitermachen, bis Ihre run()-Methode fertig ist.«

Das Geheimrezept lautet in fast allen Fällen: *schlafen*. Genau, *schlafen*. Wenn Sie einen gerade laufenden Thread schlafen legen, und sei es auch nur für ein paar Millisekunden, zwingen Sie ihn dazu, den Zustand »laufend« zu verlassen, was einem anderen Thread die Chance zur Ausführung gibt. Bei der Thread-Methode sleep() gibt es *eine* Garantie: dass ein schlafender Thread nicht wieder zum laufenden Thread wird, bevor seine Schlafenszeit rum ist. Wenn Sie beispielsweise Ihren Thread anweisen, zwei Sekunden (2.000 Millisekunden) zu schlafen, kann der Thread auf keinen Fall wieder zum laufenden Thread werden, bevor nicht die zwei Sekunden (oder mehr) verstrichen sind.



# Ein Beispiel dafür, wie schlecht sich der Scheduler vorhersagen lässt ...

## Die Ausführung dieses Codes auf einer bestimmten Maschine ...

```
public class MeinRunnable implements Runnable {

  public void run() {
    los();
  }

  public void los() {
    tuNochMehr();
  }

  public void tuNochMehr() {
    System.out.println("oben auf dem Stack");
  }
}


class ThreadTestlauf {

  public static void main (String[] args) {

    Runnable threadJob = new MeinRunnable();
    Thread meinThread = new Thread(threadJob);

    meinThread.start();

    System.out.println("zurück in main");
  }
}
```

## ... lieferte diese Ausgabe:

```
Datei Bearbeiten Fenster Hilfe NimmMichDran
% java ThreadTestlauf
zurück in main
oben auf dem Stack
% java ThreadTestlauf
oben auf dem Stack
zurück in main
% java ThreadTestlauf
oben auf dem Stack
zurück in main
% java ThreadTestlauf
oben auf dem Stack
zurück in main
% java ThreadTestlauf
zurück in main
oben auf dem Stack
% java ThreadTestlauf
oben auf dem Stack
zurück in main
% java ThreadTestlauf
zurück in main
oben auf dem Stack
```

Beachten Sie, dass die Reihenfolge zufällig wechselt. Manchmal ist der neue Thread als Erster fertig und manchmal der main-Thread.

# Wie konnten wir unterschiedliche Ergebnisse erhalten?

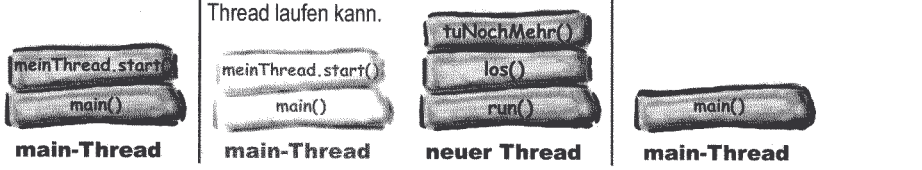
## Manchmal läuft es so:

main() startet den neuen Thread.

Der Scheduler versetzt den main-Thread aus dem Zustand »laufend« zurück in den Zustand »lauffähig«, so dass der neue Thread laufen kann.

Der Scheduler lässt den neuen Thread vollständig zu Ende laufen, so dass »oben auf dem Stack« ausgegeben wird.

Der neue Thread verschwindet, weil seine run()-Methode fertig ist. Der main-Thread wird wieder zum laufenden Thread und gibt »zurück in main« aus.




Zeit

## Und manchmal läuft es so:

main() startet den neuen Thread.

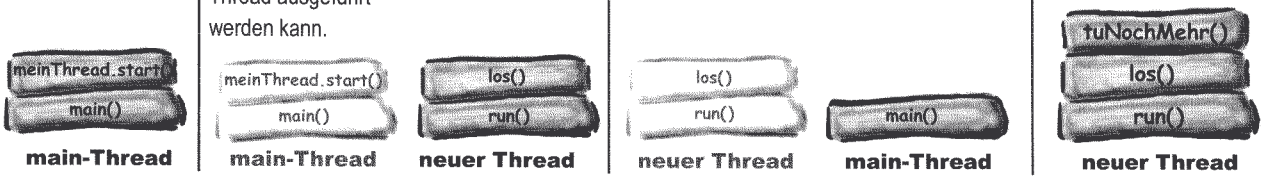
Der Scheduler versetzt den main-Thread aus dem Zustand »laufend« zurück in den Zustand »lauffähig«, so dass der neue Thread ausgeführt werden kann.

Der Scheduler lässt den neuen Thread ein Weilchen laufen, aber nicht lange genug, um die run()-Methode zu Ende auszuführen.

Der Scheduler versetzt den neuen Thread zurück in den Zustand »lauffähig«.

Der Scheduler nimmt wieder den main-Thread als laufenden Thread an die Reihe. main() gibt »zurück in main« aus.

Der neue Thread kehrt in den Zustand »laufend« zurück und gibt »oben auf dem Stack« aus.




Zeit

## Es gibt keine Dummen Fragen

**F: Ich habe Beispiele gesehen, die keine separate Runnable-Implementierung verwenden, sondern einfach eine Unterklasse von Thread erzeugen und die Methode run() des Threads überschreiben. Um einen neuen Thread zu erzeugen, ruft man dann den parameterlosen Konstruktor des Threads auf:**

```
Thread t = new Thread(); // kein Runnable
```

A: Ja, so *können* Sie Ihren Thread tatsächlich auch machen, aber betrachten Sie das einmal aus der OO-Perspektive. Welchen Zweck hat denn die Bildung von Unterklassen? Denken Sie daran, dass wir hier über zwei verschiedene Dinge reden – über den Thread (d.h. das Thread-Objekt) und über den Job des Threads (d.h. des Ausführungsstrangs). Aus OO-Sicht sind dies zwei völlig getrennte Aktivitäten, die in zwei verschiedene Klassen gehören. Eine Unterklasse von Thread sollten Sie nur dann bilden, wenn Sie einen neuen und spezifischeren Typ von Thread machen. Mit anderen Worten: Stellen Sie sich den Thread als Arbeitskraft vor und erweitern Sie die Thread-Klasse nur dann, wenn Sie spezifischeres Verhalten der *Arbeitskraft* benötigen. Aber wenn Sie nichts weiter als einen neuen *Job* brauchen, der von einem Thread/einer Arbeitskraft ausgeführt werden soll, implementieren Sie Runnable in einer separaten, *Job*-spezifischen (nicht *Arbeitskraft*-spezifischen) Klasse.

Dies ist eine Frage des Designs und hat nichts mit der Leistungsfähigkeit oder der Sprache zu tun. Es ist absolut zulässig, Thread zu erweitern und die Methode run() zu überschreiben, aber eine gute Idee ist das nur selten.

**F: Können Sie ein Thread-Objekt wiederverwenden? Können Sie ihm einen neuen Job geben und ihn dann durch den Aufruf von start() erneut starten?**

A: Nein. Wenn die run()-Methode eines Threads einmal beendet ist, lässt sich der Thread nie wieder starten. Tatsächlich geht der Thread zu diesem Zeitpunkt in einen Zustand über, den wir noch nicht erwähnt haben – er ist ***tot (dead).*** Im Zustand »tot« hat der Thread seine run()-Methode zu Ende ausgeführt und kann nie wieder gestartet werden. Das Thread-Objekt könnte sich zwar noch auf dem Heap befinden, als »lebendes Objekt«, auf dem Sie andere Methoden aufrufen können (falls das zweckdienlich ist), aber seine »Threadhaftigkeit« hat es für immer verloren. Mit anderen Worten: Es hat keinen separaten Aufruf-Stack mehr und ist damit nicht länger ein *Thread* im allgemeinen Sinn. Es ist einfach irgendein Objekt – wie alle anderen Objekte.

Es gibt jedoch Entwurfsmuster für die Erzeugung eines Pools von Threads, die man dann zur Ausführung verschiedener Jobs heranziehen kann. Aber das funktioniert nicht so, dass hier ein bereits toter Thread von Neuem gestartet würde.

## Punkt für Punkt

- Ein Thread im allgemeinen Sinn ist in Java ein separater Ausführungsstrang.
- Jeder Thread hat in Java seinen eigenen Aufruf-Stack.
- Ein Thread-Objekt – eine Instanz der Klasse java.lang.Thread – repräsentiert einen Ausführungsstrang.
- Ein Thread braucht einen Job. Der Job eines Threads ist eine Instanz von etwas, das das Interface Runnable implementiert.
- Das Interface Runnable enthält nur eine einzige Methode, run(). Dies ist die Methode, die an unterster Stelle des neuen Aufruf-Stacks steht. Mit anderen Worten: Es ist die erste Methode, die in dem neuen Thread ausgeführt wird.
- Um einen neuen Thread in Gang zu setzen, brauchen Sie ein Runnable, das Sie dem Konstruktor des Threads übergeben.
- Ein Thread ist im Zustand NEU (NEW), wenn Sie ein Thread-Objekt instantiiert, aber noch nicht start() aufgerufen haben.
- Wenn Sie einen Thread starten (durch Aufrufen der Methode start() des Thread-Objekts), wird ein neuer Stack erzeugt, an dessen unterster Stelle die run()-Methode des Runnable-Objekts steht. Der Thread ist jetzt im Zustand LAUFFÄHIG (RUNNABLE) und wartet darauf, für die Ausführung ausgewählt zu werden.
- Ein Thread wird als LAUFEND (RUNNING) bezeichnet, wenn der JVM-Thread-Scheduler ihn als laufenden Thread ausgewählt hat. Auf einer Einprozessormaschine kann es immer nur einen gerade laufenden Thread geben.
- Manchmal kann ein Thread vom Zustand LAUFEND in den Zustand BLOCKIERT (BLOCKED) versetzt werden, d.h., er ist vorübergehend nicht lauffähig. Ein Thread kann z.B. blockiert sein, weil er auf Daten aus einem Strom wartet oder weil er schlafen gegangen ist oder weil er auf die Sperre eines Objekts wartet.
- Es wird nicht garantiert, dass der Thread-Scheduler auf eine bestimmte Art und Weise arbeitet; daher können Sie nicht sicher sein, dass Threads sich abwechseln. Sie können dies jedoch unterstützen, indem Sie Ihre Threads regelmäßig schlafen schicken.

# Einen Thread schlafen legen

Eine der besten Möglichkeiten, Threads abwechselnd drankommen zu lassen, ist, sie regelmäßig schlafen zu schicken. Dazu müssen Sie nur die statische Methode sleep() aufrufen und Ihr die Schlafdauer (in Millisekunden) übergeben.

Ein Beispiel:

```
Thread.sleep(2000);
```

Dies nimmt einen Thread aus dem Zustand »laufend« heraus und hält ihn zwei Sekunden lang davon ab, wieder in den Zustand »lauffähig« zurückzukehren. Er kann *nicht* wieder zum laufenden Thread werden, bevor nicht mindestens zwei Sekunden vergangen sind.

Etwas bedauerlich ist, dass die sleep-Methode eine InterruptedException auslöst, eine geprüfte Exception. Alle Aufrufe von sleep müssen daher mit einem try/catch-Block umgeben werden (oder die Exception muss deklariert werden). Ein Aufruf von sleep sieht daher in der Realität so aus:

```
try {
  Thread.sleep(2000);
} catch (InterruptedException ex) {
  ex.printStackTrace();
}
```

Es ist allerdings unwahrscheinlich, dass jemals etwas Ihren Thread beim Schlafen unterbrechen wird; die Exception steht in der API, um einen Thread-Kommunikationsmechanismus zu unterstützen, den im wirklichen Leben fast niemand verwendet. Trotzdem müssen Sie sich an das Gesetz vom Behandeln oder Deklarieren halten – gewöhnen Sie sich einfach daran, Ihre sleep()-Aufrufe mit try/catch zu umgeben.

Sie wissen nun, dass Ihr Thread nicht aufwachen wird, *bevor* die angegebene Zeit verstrichen ist, aber ist es möglich, dass er erst eine Weile *nach* dem Ablaufen des »Weckers« aufwacht? Die Antwort ist: ja und nein. Es ist eigentlich auch egal, denn wenn der Thread aufwacht, ***geht er immer automatisch in den Zustand »lauffähig«!*** Er wacht nicht automatisch zur festgesetzten Zeit auf und wird sofort zum laufenden Thread. Wenn ein Thread aufwacht, ist er wieder einmal der Gnade des Thread-Schedulers ausgeliefert. Bei Programmen, die kein perfektes Timing erfordern und nur wenige Threads haben, kann es zwar so aussehen, als ob der Thread aufwacht und genau nach Plan direkt weiter ausgeführt wird (z.B. nach 2.000 Millisekunden). Aber verwetten Sie nicht Ihr Programm darauf!

**Legen Sie Ihren Thread schlafen, wenn Sie sicher sein wollen, dass auch andere Threads eine Chance zum Laufen erhalten.**

**Wenn der Thread aufwacht, befindet er sich immer zunächst im Zustand »lauffähig« und wartet darauf, dass ihn der Thread-Scheduler wieder für die Ausführung aussucht.**

# Wie wir unser Programm mit sleep besser vorhersagbar machen

Erinnern Sie sich an eines unserer vorherigen Beispiele, in dem wir bei jeder Ausführung ein anderes Ergebnis erhielten? Sehen Sie sich noch einmal dessen Quellcode und die Ausgabe an. Manchmal musste main warten, bis der neue Thread fertig war (und »oben auf dem Stack« ausgab), während ein anderes Mal der neue Thread wieder in den Zustand »lauffähig« versetzt wurde, bevor er fertig war, so dass der main-Thread wiederkommen und »zurück in main« ausgeben konnte. Wie können wir das in Ordnung bringen? Beantworten Sie vorher erst einmal folgende Frage: Wo können Sie einen Aufruf von sleep() hinsetzen, um sicherzugehen, dass »zurück in main« immer vor »oben auf dem Stack« ausgegeben wird?

Wir warten so lange, bis Sie sich die Antwort überlegt haben (es gibt mehr als eine richtige Antwort).

Und? Wissen Sie schon, wo?

Das hier wollen wir haben – eine einheitliche Reihenfolge der ausgegebenen Zeilen:

```
Datei Bearbeiten Fenster Hilfe Schlummertaste
% java ThreadTestlauf
zurück in main
oben auf dem Stack
% java ThreadTestlauf
zurück in main
oben auf dem Stack
% java ThreadTestlauf
zurück in main
oben auf dem Stack
% java ThreadTestlauf
zurück in main
oben auf dem Stack
% java ThreadTestlauf
zurück in main
oben auf dem Stack
```

```
public class MeinRunnable implements Runnable {

  public void run() {
    los();
  }

  public void los() {

    try {
      Thread.sleep(2000);
    } catch(InterruptedException ex) {
      ex.printStackTrace();
    }

    tuNochMehr();
  }

  public void tuNochMehr() {
    System.out.println("oben auf dem Stack");
  }
}

class ThreadTestlauf {
  public static void main (String[] args) {
    Runnable threadJob = new MeinRunnable();
    Thread meinThread = new Thread(threadJob);
    meinThread.start();
    System.out.println("zurück in main");
  }
}
```

Wenn Sie hier sleep aufrufen, zwingt das den neuen Thread, den Zustand >>laufend<< zu verlassen.

Der main-Thread wird wieder zum laufenden Thread und gibt >>zurück in main<< aus. Dann folgt eine (etwa zwei Sekunden dauernde) Pause, bevor wir zu dieser Zeile kommen, die tuNochMehr() aufruft und >>oben auf dem Stack<< ausgibt.

# Erzeugen und Starten von *zwei* Threads

Threads haben Namen. Sie können Ihre Threads auf einen Namen Ihrer Wahl taufen oder ihre vorgegebenen Namen akzeptieren. Das Tolle an den Namen ist, dass Sie damit feststellen können, welcher Thread gerade läuft. Das folgende Beispiel startet zwei Threads. Jeder Thread hat die gleiche Aufgabe: in einer Schleife zu laufen und bei jeder Iteration den Namen des aktuell laufenden Threads auszugeben.

```
public class ZweiThreads implements Runnable {

  public static void main(String[] args) {
    ZweiThreads aufgabe = new ZweiThreads();
    Thread alpha = new Thread(aufgabe);
    Thread beta = new Thread(aufgabe);
    alpha.setName("Alpha-Thread");
    beta.setName("Beta-Thread");
    alpha.start();
    beta.start();
  }
  public void run() {
    for (int i = 0; i < 25; i++) {
      String threadName = Thread.currentThread().getName();
      System.out.println("jetzt läuft der " + threadName);
    }
  }
}
```

Machen Sie eine Instanz von Runnable.

Machen Sie zwei Threads mit dem gleichen Runnable (also dem gleichen Job – in ein paar Seiten sagen wir noch mehr zum Thema >>zwei Threads mit nur einem Runnable<<).

Geben Sie den Threads Namen.

Starten Sie die Threads.

Jeder Thread durchläuft diese Schleife und gibt bei jedem Durchgang seinen Namen aus.

# Was wird passieren?

Teil der Ausgabe bei 25facher Wiederholung der Schleife.

```
Datei Bearbeiten Fenster Hilfe Centauri
jetzt läuft der Alpha-Thread
jetzt läuft der Alpha-Thread
jetzt läuft der Alpha-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Alpha-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Beta-Thread
jetzt läuft der Alpha-Thread
```

Werden die Threads sich abwechseln? Werden Sie mal diesen, mal jenen Thread-Namen sehen? Wie oft wird es wechseln? Bei jedem Schleifendurchlauf? Alle fünf Durchläufe?

Sie kennen die Antwort bereits: *Wir wissen es nicht!* Es hängt vom Scheduler ab. Und auf Ihrem Betriebssystem, mit Ihrer speziellen JVM, auf Ihrer CPU erhalten Sie vielleicht ein ganz anderes Ergebnis als wir.

Mit OS X 10.2 (Jaguar) wird bei fünf oder weniger Iterationen erst der Alpha-Thread zu Ende ausgeführt und dann der Beta-Thread. Sehr einheitlich. Nicht 100%ig, aber sehr einheitlich.

Wenn Sie jedoch die Schleife auf 25 oder mehr Durchläufe setzen, kommen die Dinge ins Wackeln. Dann schafft der Alpha-Thread vielleicht nicht alle 25 Iterationen, bevor ihn der Scheduler wieder in den Zustand »lauffähig« zurückschiebt, um dem Beta-Thread eine Chance zu geben.

## Leider doch. Es GIBT eine Kehrseite. Threads können zu Nebenläufigkeitsproblemen führen.

Nebenläufigkeitsprobleme führen zu Konkurrenzsituationen. Konkurrenzsituationen führen zu Datenverfälschung. Verfälschte Daten lösen Ängste aus ... den Rest kennen Sie ja.

Es läuft alles auf das folgende, potenziell tödliche Szenario hinaus: Zwei oder mehr Threads haben Zugriff auf die *Daten* desselben Objekts. Mit anderen Worten, Methoden auf zwei verschiedenen Stacks rufen beide z.B. Getter und Setter ein und desselben Objekts auf dem Heap auf.

Es ist eine »Die-linke-Hand-weiß-nicht-was-die-rechte-tut«-Geschichte. Zwei Threads, sorgenfrei und unbeschwert, führen fröhlich pfeifend ihre Methoden aus, und jeder von ihnen glaubt, er sei »der einzig wahre Thread«. Der einzige, der zählt. Denn ein Thread, der gerade nicht läuft, sondern im Zustand »lauffähig« (oder »blockiert«) ist, ist ja praktisch bewusstlos. Wenn er wieder zum laufenden Thread wird, weiß er gar nicht, dass er jemals angehalten hat.

# Ehe in Gefahr

## Ist dieses Paar noch zu retten?

***Gleich – in einer Sondersendung aus unserer Reihe »Fragen Sie Dr. Winter«!***

[Abschrift von Folge 42]

Herzlich willkommen in unserer Sendung!

In unserem aktuellen Thema geht es um die beiden wichtigsten Gründe, warum Paare sich trennen – Finanz- und Schlafprobleme.

Rainer und Monika: Opfer des »Zwei-Leute-ein-Konto«-Problems.

Unser heutiges Problempaar, Rainer und Monika, teilt Bankkonto und Bett. Allerdings nicht mehr lange, wenn wir nicht eine Lösung finden können. Das Problem? Die klassische »Zwei-Leute-ein-Bankkonto«-Geschichte.

Und so hat Monika es mir beschrieben:

»Rainer und ich haben vereinbart, dass keiner von uns das Konto überziehen wird. Wir machen es also so: Wer von uns etwas abheben will, muss den Kontostand überprüfen, *bevor* er die Abhebung durchführt. Es schien alles so einfach. Aber plötzlich werden unsere Schecks gesperrt, und wir zahlen jede Menge Überziehungszinsen!

Ich dachte, das wäre unmöglich, unser Verfahren wäre sicher. Aber dann passierte *das hier*:

Rainer brauchte 50 €, also hat er den Kontostand geprüft und gesehen, dass noch 100 € drauf waren. Kein Problem also, er will den Betrag abheben. **Aber dann schläft er erst einmal ein!**

Rainer schläft ein, nachdem er den Kontostand geprüft hat, aber bevor er das Geld abgehoben hat. Wenn er aufwacht, hebt er es sofort ab, ohne sich den Kontostand noch einmal anzusehen.

Und jetzt komme ich, während Rainer noch schläft, und will eine Abhebung von 100 € machen. Ich prüfe den Kontostand; es sind 100 € drauf (weil Rainer noch schläft und seinen Betrag noch nicht abgehoben hat), also denke ich, kein Problem. Ich hebe das Geld ab, und da ist noch alles okay. Aber dann wacht Rainer auf, führt *seine* Abhebung zu Ende, und plötzlich ist unser Konto überzogen! Er wusste noch nicht mal, dass er eingeschlafen war, also hat er einfach weitergemacht und seinen Vorgang zu Ende geführt, ohne noch einmal den Kontostand zu überprüfen. Sie müssen uns helfen, Dr. Winter!«

Gibt es eine Lösung? Oder ist die Ehe dem Untergang geweiht? Wir können Rainer nicht am Einschlafen hindern, aber können wir vielleicht sicherstellen, dass Monika nicht auf das Konto zugreifen kann, bevor er aufgewacht ist?

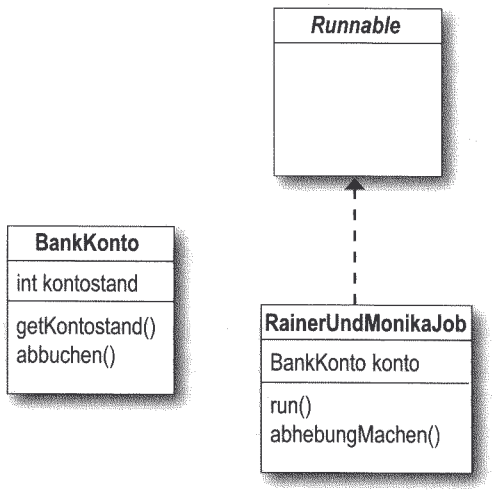
Nehmen Sie sich einen Moment Zeit und denken Sie darüber nach, während wir eine Werbepause machen.

# Das Rainer-und-Monika-Problem in Codeform

Das folgende Beispiel zeigt, was passieren kann, wenn zwei Threads (Rainer und Monika) ein Objekt (das Bankkonto) miteinander teilen.

Der Code hat zwei Klassen, BankKonto und RainerUndMonikaJob. Die Klasse RainerUndMonikaJob implementiert Runnable und repräsentiert das Verhalten, das sowohl Rainer als auch Monika haben: den Kontostand prüfen und die Abhebung durchführen. Aber natürlich schläft jeder Thread zwischen dem Überprüfen des Kontostands und dem eigentlichen Abheben ein.

Die Klasse RainerUndMonikaJob hat eine Instanzvariable vom Typ BankKonto, die das gemeinsame Konto repräsentiert.




**1 Eine Instanz von RainerUndMonikaJob machen.**

Das RainerUndMonikaJob-Objekt ist das Runnable (der Job), und da Monika und Rainer das Gleiche tun (Kontostand prüfen und Geld abheben), brauchen wir nur eine einzige Instanz.

```
RainerUndMonikaJob derJob = new RainerUndMonikaJob();
```

**2 Zwei Threads mit dem gleichen Runnable machen** (der RainerUndMonikaJob-Instanz).

```
Thread eins = new Thread(derJob);
Thread zwei = new Thread(derJob);
```

**3 Den Threads einen Namen geben und sie starten.**

```
eins.setName("Rainer");
zwei.setName("Monika");
eins.start();
zwei.start();
```

**4 Beide Threads bei der Ausführung der run()-Methode beobachten** (bei der Überprüfung des Kontostands und dem eigentlichen Abheben).

Ein Thread repräsentiert Rainer, der andere repräsentiert Monika. Beide Threads prüfen immer wieder den Kontostand und heben dann etwas ab - aber nur, wenn genug da ist!

```
if (konto.getKontostand() >= betrag) {
  try {
    Thread.sleep(500);
  } catch(InterruptedException ex) {ex.printStackTrace(); }
}
```

**In der run()-Methode machen wir genau das, was auch Monika und Rainer tun würden – den Kontostand prüfen und Geld abheben, wenn genug auf dem Konto ist.**

**Dies sollte vor der Überziehung des Kontos schützen.**

**Außer ... wenn Monika und Rainer jedes Mal einschlafen, nachdem sie den Kontostand geprüft haben, aber bevor sie die Abhebung zu Ende führen.**

# Das Rainer-und-Monika-Beispiel

```
class BankKonto {
  private int kontostand = 100;

  public int getKontostand() {
    return kontostand;
  }

  public void abbuchen(int betrag) {
    kontostand = kontostand - betrag;
  }
}

public class RainerUndMonikaJob implements Runnable {

  private BankKonto konto = new BankKonto();

  public static void main (String [] args) {
    RainerUndMonikaJob derJob = new RainerUndMonikaJob();
    Thread eins = new Thread(derJob);
    Thread zwei = new Thread(derJob);
    eins.setName("Rainer");
    zwei.setName("Monika");
    eins.start();
    zwei.start();
  }

  public void run() {
    for (int x = 0; x < 10; x++) {
      abhebungMachen(10);
      if (konto.getKontostand() < 0) {
        System.out.println("Überzogen!");
      }
    }
  }

  private void abhebungMachen(int betrag) {
    if (konto.getKontostand() >= betrag) {
      System.out.println(Thread.currentThread().getName() + " ist im Begriff abzuheben.");
      try {
        System.out.println(Thread.currentThread().getName() + " schläft ein.");
        Thread.sleep(500);
      } catch(InterruptedException ex) {ex.printStackTrace(); }
      System.out.println(Thread.currentThread().getName() + " ist aufgewacht.");
      konto.abbuchen(betrag);
      System.out.println(Thread.currentThread().getName() + " führt die Abhebung zu Ende.");
      }
    else {
      System.out.println("Leider nicht genug Geld für " + Thread.currentThread().getName());
    }
  }
}
```

Zu Beginn beträgt der Kontostand 100 €.

Es gibt nur EINE Instanz des RainerUndMonika-Job, d.h. nur EINE Instanz des Bankkontos. Beide Threads werden auf dieses eine Konto zugreifen.

Instantiieren Sie das Runnable (den Job).

Machen Sie zwei Threads und übergeben Sie beiden den gleichen Runnable-Job. Das bedeutet, dass beide Threads auf ein und dieselbe Instanzvariable konto in der Runnable-Klasse zugreifen.

In der run()-Methode durchläuft ein Thread die Schleife und versucht bei jeder Iteration, eine Abhebung zu machen. Nach dem Abheben prüft er den Kontostand erneut, um zu sehen, ob das Konto überzogen ist.

Prüfen Sie den Kontostand, und wenn nicht genug Geld da ist, wird einfach eine Meldung ausgegeben. WENN genug da ist, schlafen wir ein, wachen dann wieder auf und führen die Abhebung zu Ende, genau wie Rainer es gemacht hat.

Wir haben eine ganze Menge print-Anweisungen eingefügt, damit wir bei der Ausführung sehen, was passiert.

## Ausgabe von Rainer und Monika

```
Datei Bearbeiten Fenster Hilfe KreditKarte
Rainer ist im Begriff abzuheben.
Rainer schläft ein.
Monika ist aufgewacht.
Monika führt die Abhebung zu Ende.
Monika ist im Begriff abzuheben.
Monika schläft ein.
Rainer ist aufgewacht.
Rainer führt die Abhebung zu Ende.
Rainer ist im Begriff abzuheben.
Rainer schläft ein.
Monika ist aufgewacht.
Monika führt die Abhebung zu Ende.
Monika ist im Begriff abzuheben.
Monika schläft ein.
Rainer ist aufgewacht.
Rainer führt die Abhebung zu Ende.
Rainer ist im Begriff abzuheben.
Rainer schläft ein.
Monika ist aufgewacht.
Monika führt die Abhebung zu Ende.
Leider nicht genug Geld für Monika
Leider nicht genug Geld für Monika
Leider nicht genug Geld für Monika
Leider nicht genug Geld für Monika
Leider nicht genug Geld für Monika
Rainer ist aufgewacht.
Rainer führt die Abhebung zu Ende.
Überzogen!
Leider nicht genug Geld für Rainer
Überzogen!
Leider nicht genug Geld für Rainer
Überzogen!
Leider nicht genug Geld für Rainer
Überzogen!
```

Wie konnte das passieren?

**Die Methode abhebungMachen() überprüft jedes Mal den Kontostand, bevor die eigentliche Abhebung durchgeführt wird – und trotzdem überziehen wir das Konto.**

### Hier ein mögliches Szenario:

Rainer prüft den Kontostand, sieht, dass genug Geld da ist, und schläft dann ein.

In der Zwischenzeit kommt Monika und überprüft den Kontostand. Sie sieht ebenfalls, dass genug Geld da ist, und hat keine Ahnung, dass Rainer gleich aufwachen wird und eine Abhebung zu Ende führt.

Monika schläft ein.

Rainer wacht auf und führt seine Abhebung zu Ende.

Monika wacht auf und führt ihre Abhebung zu Ende. Und da haben wir den Salat! In der Zwischenzeit – nachdem sie den Kontostand geprüft und bevor sie die Abhebung zu Ende geführt hatte – war Rainer aufgewacht und hatte Geld vom Konto genommen.

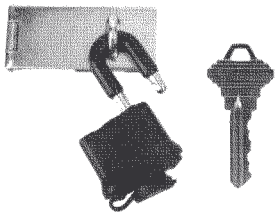
**Monikas Überprüfung des Kontostands war nicht gültig, weil Rainer ihn bereits überprüft hatte und noch mitten in einem Abhebungsvorgang steckte.**

Monika darf nicht an das Konto gehen können, bevor Rainer aufwacht und seine Transaktion zu Ende führt. Und umgekehrt.

# Sie brauchen ein Schloss für den Kontozugang!

**Das Schloss funktioniert so:**

① Zu der Kontotransaktion (Kontostand prüfen und Geld abheben) gibt es ein Schloss. Es existiert nur *ein* Schlüssel, und der hängt neben dem Schloss, bis jemand auf das Konto zugreifen will.



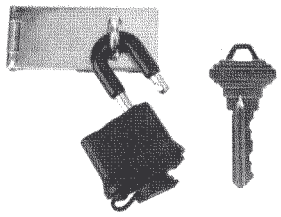
**Die Bankkonto-Transaktion ist unversperrt, wenn gerade niemand auf das Konto zugreift.**

② Wenn Rainer auf das Konto zugreifen will (um den Kontostand zu prüfen und Geld abzuheben), schließt er ab und steckt den Schlüssel in seine Hosentasche. Jetzt kann niemand anderes auf das Konto zugreifen, weil der Schlüssel weg ist.

**Wenn Rainer auf das Konto zugreifen will, versperrt er das Schloss und nimmt den Schlüssel an sich.**

③ **Rainer behält den Schlüssel in seiner Tasche, bis er mit der Transaktion fertig ist.** Er hat den einzigen Schlüssel, daher kann Monika nicht auf das Konto (oder das Scheckbuch) zugreifen, bevor Rainer wieder aufgeschlossen und den Schlüssel zurückgegeben hat.

Wenn Rainer jetzt einschläft, nachdem er das Konto geprüft hat, hat er die Gewissheit, dass der Kontostand bei seinem Aufwachen noch der gleiche ist, weil er den Schlüssel behalten hat, während er schlief.



**Wenn Rainer fertig ist, sperrt er das Schloss auf und hängt den Schlüssel zurück. Jetzt ist der Schlüssel für Monika (oder erneut für Rainer) verfügbar und gestattet den Zugriff auf das Konto.**

# Wir müssen die Methode abhebungMachen() als unteilbare atomare Einheit ausführen.

Wir müssen dafür sorgen, dass ein Thread, sobald er einmal in die Methode abhebungMachen() eingetreten ist, *diese Methode zu Ende ausführen darf,* bevor irgendein anderer Thread hineindarf.

Mit anderen Worten: Wir müssen sicherstellen, dass einem Thread, der den Kontostand geprüft hat, garantiert wird, dass er aufwachen und die Abhebung zu Ende ausführen kann, *bevor irgendein anderer Thread den Kontostand prüfen kann!*

Mit dem Schlüsselwort **synchronized** können Sie eine Methode so modifizieren, dass immer nur ein Thread auf einmal auf sie zugreifen kann.

Und so schützen Sie das Konto! Sie schließen nicht das Bankkonto selbst ab, sondern Sie schließen die Methode für die Kontotransaktion ab. Auf diese Weise kann ein Thread die gesamte Transaktion von Anfang bis Ende durchführen, selbst wenn er mitten in der Methode einschläft!

Wenn man also nicht das Bankkonto abschließt, was genau *wird* dann abgeschlossen? Das Runnable-Objekt? Der Thread selbst?

Das sehen wir uns auf der nächsten Seite an. Im Code ist es allerdings ganz einfach – fügen Sie in Ihrer Methodendeklaration einfach den Modifizierer `synchronized` ein:

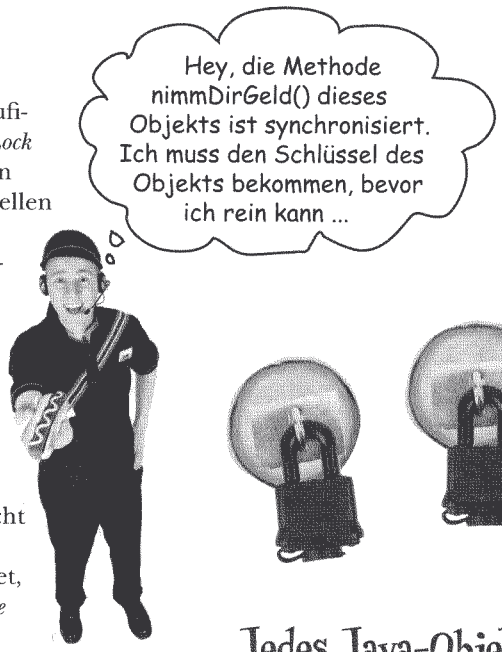
**Das Schlüsselwort `synchronized` bedeutet, dass ein Thread einen Schlüssel braucht, um auf den synchronisierten Code zugreifen zu können.**

**Um Ihre *Daten* (z.B. das Bankkonto) zu schützen, synchronisieren Sie die *Methoden*, die auf diese Daten einwirken.**

```
private synchronized void abhebungMachen(int betrag) {

  if (konto.getKontostand() >= betrag) {
    System.out.println(Thread.currentThread().getName() + " ist im Begriff abzuheben.");
    try {
      System.out.println(Thread.currentThread().getName() + " schläft ein.");
      Thread.sleep(500);
    } catch(InterruptedException ex) {ex.printStackTrace(); }
    System.out.println(Thread.currentThread().getName() + " ist aufgewacht.");
    konto.abbuchen(betrag);
    System.out.println(Thread.currentThread().getName() + " führt die Abhebung zu Ende.");
  } else {
    System.out.println("Leider nicht genug Geld für " + Thread.currentThread().getName());
  }
}
```

(Anmerkung für Leser, die Ahnung von Physik haben: Ja, die Konvention für die Verwendung des Worts >>atomar<< in diesem Zusammenhang passt nicht zu der ganzen Sache mit den subatomaren Teilchen. Denken Sie an Newton und nicht an Einstein, wenn Sie das Wort >>atomar<< im Zusammenhang mit Threads oder Transaktionen hören. Diese Konvention stammt ja nicht von UNS. Wenn WIR zuständig wären, würden wir auf praktisch alles, was mit Threads zu tun hat, die Heisenbergsche Unschärferelation anwenden.)

# Das Schloss eines Objekts verwenden

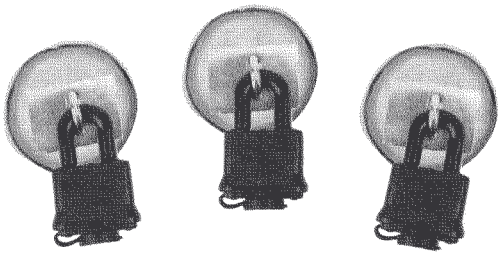
Jedes Objekt hat ein Schloss (im Deutschen häufiger als *Sperre* oder mit dem englischen Begriff *Lock* bezeichnet). Meistens ist das Schloss offen, dann können Sie sich einen daneben liegenden virtuellen Schlüssel dazudenken. Objektsperren kommen nur dann ins Spiel, wenn es synchronisierte Methoden gibt. Hat ein Objekt eine oder mehrere synchronisierte Methoden, ***kann ein Thread erst dann in eine synchronisierte Methode eintreten, wenn er den Schlüssel zum Schloss des Objekts erhält!***

Die Sperren gehören nicht zu einer *Methode*, sondern zum *Objekt*. Wenn ein Objekt zwei synchronisierte Methoden hat, bedeutet das nicht einfach, dass zwei Threads nicht gleichzeitig in dieselbe Methode eintreten können. Es bedeutet, dass zwei Threads nicht gleichzeitig in *irgendeine* der Methoden eintreten können.

Überlegen Sie mal: Wenn Sie mehrere Methoden haben, die auf die Instanzvariablen eines Objekts einwirken können, müssen alle diese Methoden durch »synchronized« geschützt werden.

Das Ziel der Synchronisierung ist der Schutz von kritischen Daten. Aber denken Sie daran, Sie verschließen nicht die Daten selbst, sondern synchronisieren die Methoden, die auf die Daten *zugreifen*.

Was passiert also, wenn ein Thread sich durch seinen Aufruf-Stack ackert (beginnend mit der run()-Methode) und dann plötzlich auf eine synchronisierte Methode stößt? Er erkennt, dass er einen Schlüssel für das Objekt braucht, bevor er Zutritt zu der Methode erhält. Er sieht sich nach dem Schlüssel um (um all das kümmert sich die JVM; es gibt in Java keine API, mit der man auf Objektschlösser zugreifen kann), und sobald dieser verfügbar ist, schnappt er ihn sich und tritt in die Methode ein.

Von diesem Zeitpunkt an klammert sich der Thread an diesen Schlüssel, als hinge sein Leben davon ab. Er lässt ihn nicht los, bevor er nicht mit der synchronisierten Methode fertig ist. Während dieser Thread also den Schlüssel festhält, erhält kein anderer Thread Zutritt zu *irgendeiner* der synchronisierten Methoden in diesem Objekt, weil der einzige Schlüssel für dieses Objekt nicht zur Verfügung steht.



Jedes Java-Objekt hat ein Schloss. Ein Schloss hat nur einen Schlüssel.

Meistens ist das Schloss unversperrt, und keiner interessiert sich dafür.

Aber wenn ein Objekt synchronisierte Methoden enthält, kann ein Thread nur dann in eine der synchronisierten Methoden eintreten, wenn der Schlüssel für das Schloss des Objekts zur Verfügung steht. Mit anderen Worten: nur wenn sich nicht bereits ein anderer Thread den einzigen Schlüssel geschnappt hat.

# Das gefürchtete »Problem der verlorenen Aktualisierung«

Hier kommt ein weiteres klassisches Nebenläufigkeitsproblem, das aus der Datenbankwelt stammt. Es ist zwar eng mit der Rainer-und-Monika-Story verwandt, aber wir nehmen hier dieses Beispiel, um einige weitere Punkte aufzuzeigen.

Bei der verlorenen Aktualisierung (lost update) geht es um einen bestimmten Vorgang:

Schritt 1: Kontostand abfragen

```
int i = kontostand;
```

Schritt 2: Kontostand um 1 erhöhen

```
kontostand = i + 1;
```

Um das Problem aufzuzeigen, wenden wir einen Trick an: Wir zwingen den Computer, die Veränderung des Kontostands in zwei Schritten vorzunehmen. Im wahren Leben würde dafür eine einzige Anweisung genügen: `kontostand++;`

Aber dadurch, dass wir das in *zwei* Schritte aufteilen, wird das Problem eines nicht atomaren Vorgangs klar. Stellen Sie sich also vor, die zwei (oder mehr) Schritte dieser Methode bestünden nicht aus etwas so Trivialem wie »Kontostand abfragen und dann 1 zum aktuellen Stand addieren«, sondern sie wären komplexer und ließen sich nicht in einer Anweisung zusammenfassen.

Beim Problem der »verlorenen Aktualisierung« haben wir zwei Threads, die beide versuchen, den Kontostand zu erhöhen.

```
class TestSync implements Runnable {

  private int kontostand;

  public void run() {
    for(int i = 0; i < 50; i++) {
      inkrementieren();
      System.out.println("kontostand beträgt " + kontostand);
    }
  }

  public void inkrementieren() {
    int i = kontostand;
    kontostand = i + 1;
  }
}

public class TestSyncTest {
  public static void main (String[] args) {
  TestSync job = new TestSync();
  Thread a = new Thread(job);
  Thread b = new Thread(job);
  a.start();
  b.start();
  }
}
```

Jeder Thread läuft 50-mal und erhöht bei jedem Durchlauf den Kontostand.

Hier ist die kritische Stelle! Wir erhöhen den Kontostand, indem wir 1 zu dem Wert des Kontostands, den er ZUM ZEITPUNKT UNSERER ABFRAGE hatte, addieren (anstatt 1 zum AKTUELLEN Wert zu addieren).

# Führen wir diesen Code aus ...

① Thread A läuft eine Weile

Setzen Sie den Wert des Kontostands in die Variable i.
Der Kontostand ist 0, also ist i jetzt gleich 0.
Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.
Der Kontostand ist jetzt 1.
Setzen Sie den Wert des Kontostands in die Variable i.
Der Kontostand ist 1, also ist i jetzt gleich 1.
Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.
Der Kontostand ist jetzt 2.

② Thread B läuft eine Weile

Setzen Sie den Wert des Kontostands in die Variable i.
Der Kontostand ist 2, also ist i jetzt gleich 2.
Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.
Der Kontostand ist jetzt 3.
Setzen Sie den Wert des Kontostands in die Variable i.
Der Kontostand ist 3, also ist i jetzt gleich 3.

*[Thread B wird jetzt wieder in den Zustand »lauffähig« zurückversetzt, **bevor** er den Wert des Kontostands auf 4 setzt.]*

③ Thread A läuft weiter und fährt da fort, wo er aufgehört hat

Setzen Sie den Wert des Kontostands in die Variable i.
Der Kontostand ist 3, also ist i jetzt gleich 3.
Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.
Der Kontostand ist jetzt 4.
Setzen Sie den Wert des Kontostands in die Variable i.
Der Kontostand ist 4, also ist i jetzt gleich 4.
Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.
Der Kontostand ist jetzt 5.

④ Thread B läuft weiter und fährt genau da fort, wo er aufgehört hat!

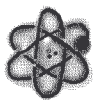
Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.

**Der Kontostand ist jetzt 4.** ← Oje!!

Thread A hat den Kontostand auf 5 erhöht, aber nun kommt Thread B zurück und macht die Aktualisierung von A zunichte, als hätte sie nie stattgefunden.

**Wir haben die letzten Aktualisierungen verloren, die Thread A gemacht hat! Thread B hatte vorher schon den Wert des Kontostands »ausgelesen«, und als B aufgewacht ist, hat er einfach weitergemacht, als hätte er nie etwas verpasst.**

# Machen Sie die Methode inkrementieren() atomar. Synchronisieren Sie sie!



Eine Synchronisierung der Methode inkrementieren() löst das »Problem der verlorenen Aktualisierung«, da sie die beiden Schritte in der Methode zu einer unteilbaren Einheit zusammenschweißt.

```
public synchronized void inkrementieren() {
  int i = kontostand;
  kontostand = i + 1;
}
```

**Wenn ein Thread einmal Zutritt zu der Methode erhalten hat, müssen wir dafür sorgen, dass alle Schritte in der Methode zu Ende ausgeführt werden (als ein einziger atomarer Prozess), bevor irgendein anderer Thread in die Methode eintreten kann.**

---

## Es gibt keine Dummen Fragen

**F: Klingt, als wäre es eine gute Idee, alles zu synchronisieren – einfach, damit es threadsicher ist.**

A: Nein, das ist keine gute Idee. Synchronisierung ist nicht umsonst zu haben. Erst einmal: Eine synchronisierte Methode ist mit einem gewissen Overhead verbunden. Anders ausgedrückt: Wenn Code auf eine synchronisierte Methode trifft, gibt es einen Leistungseinbruch (auch wenn Sie das normalerweise nicht merken würden), während die Frage »Ist der Schlüssel verfügbar?« geklärt wird.

Zweitens kann eine synchronisierte Methode Ihr Programm auch ausbremsen, weil Synchronisierung die Nebenläufigkeit einschränkt. Mit anderen Worten, eine synchronisierte Methode zwingt andere Threads, sich in einer Reihe aufzustellen und zu warten, bis sie dran sind. Vielleicht stellt das in Ihrem Code kein Problem dar, aber Sie müssen es zumindest bedenken.

Und drittens das, was man am meisten fürchten muss: Synchronisierte Methoden können zu einem »Deadlock« führen, d.h., sie können sich gegenseitig blockieren (vgl. Seite 516).

Eine gute Faustregel ist es, sich bei der Synchronisierung auf das notwendige Minimum zu beschränken. Und Sie können tatsächlich sogar winzige Einzelheiten synchronisieren, die noch kleiner sind als eine Methode. Wir brauchen das in diesem Buch zwar nicht, aber Sie können mit dem Schlüsselwort synchronized auch auf der Ebene von ein oder mehreren Anweisungen synchronisieren statt auf der Ebene ganzer Methoden.

tuWas() muss nicht synchronisiert sein, daher synchronisieren wir nicht die gesamte Methode los().

```
public void los() {
  tuWas();

  synchronized(this) {
    tuWasKritisches();
    tuWasNochKritischeres();
  }
}
```

Hier werden nur diese beiden Methodenaufrufe zu einer atomaren Einheit zusammengefasst. Wenn Sie das Schlüsselwort synchronized INNERHALB einer Methode verwenden statt in einer Methodendeklaration, müssen Sie als Argument ein Objekt liefern, dessen Schlüssel sich der Thread holen muss.

Auch wenn es noch andere Möglichkeiten gibt, synchronisiert man doch fast immer auf dem aktuellen Objekt (this). Das ist dasselbe Objekt, dessen Schloss Sie bei einer Synchronisierung der gesamten Methode benutzen würden.

### ① Thread A läuft eine Weile

Versuchen Sie, in die Methode inkrementieren() einzutreten.

Die Methode ist synchronisiert, also **holen Sie sich den Schlüssel** für dieses Objekt.

Setzen Sie den Wert des Kontostands in die Variable i.

Der Kontostand ist 0, also ist i jetzt gleich 0.

Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.

Der Kontostand ist jetzt 1.

**Geben Sie den Schlüssel zurück** (er ist fertig mit der Methode inkrementieren()).

Treten Sie erneut in die Methode inkrementieren() ein und **holen Sie sich den Schlüssel.**

Setzen Sie den Wert des Kontostands in die Variable i.

Der Kontostand ist 1, also ist i jetzt gleich 1.

*[Thread A wird jetzt wieder in den Zustand »lauffähig« zurückversetzt, aber weil er die synchronisierte Methode nicht zu Ende ausgeführt hat, behält er den Schlüssel.]*

### ② Thread B wird zur Ausführung ausgewählt

Versuchen Sie, in die Methode inkrementieren() einzutreten. Die Methode ist synchronisiert, also brauchen wir den Schlüssel.

**Der Schlüssel ist nicht verfügbar.**

*[Thread B wird in einen »Objektschloss-nicht-verfügbar«-Warteraum geschickt.]*

### ③ Thread A läuft weiter und fährt da fort, wo er aufgehört hat (erinnern Sie sich, er hat noch immer den Schlüssel)

Setzen Sie den Wert des Kontostands auf das Ergebnis von i + 1.

Der Kontostand ist jetzt 2.

**Geben Sie den Schlüssel zurück.**

*[Thread A wird jetzt wieder in den Zustand »lauffähig« zurückversetzt, aber da er die Methode inkrementieren() beendet hat, behält er den Schlüssel NICHT.]*

### ④ Thread B wird zur Ausführung ausgewählt

Versuchen Sie, Zutritt zur Methode inkrementieren() zu erhalten. Die Methode ist synchronisiert, also brauchen wir den Schlüssel.

Dieses Mal IST der Schlüssel verfügbar – holen Sie ihn sich!

Setzen Sie den Wert des Kontostands in die Variable i.

[und so weiter und so fort ...]

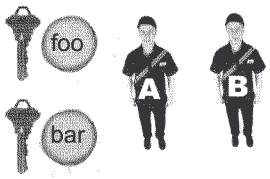
# Die tödliche Seite der Synchronisierung

Seien Sie vorsichtig, wenn Sie synchronisierten Code verwenden, denn nichts kann Ihr Programm so in die Knie zwingen wie ein Thread-Deadlock. Dabei handelt es sich um eine gegenseitige Blockierung, zu der es kommt, wenn Sie zwei Threads haben, die beide jeweils einen Schlüssel festhalten, den der andere haben möchte. Es gibt keinen Weg, der aus diesem Szenario herausführt, daher sitzen die beiden Threads einfach da und warten. Und warten. Und warten.

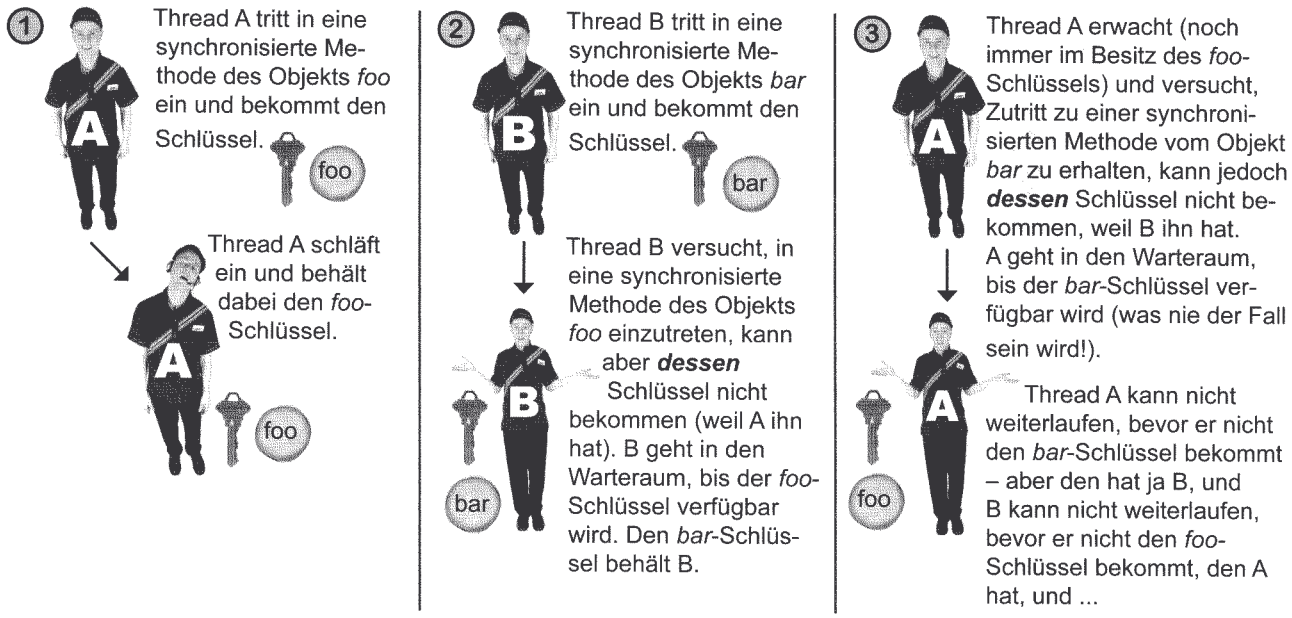
Wenn Sie mit Datenbanken oder anderen Anwendungsservern vertraut sind, kommt Ihnen das Problem vielleicht bekannt vor; Datenbanken haben oft einen ähnlichen Sperrmechanismus wie die Synchronisierung. Aber ein echtes Transaktionsmanagement-System kann Deadlocks eventuell auflösen. Es könnte zum Beispiel annehmen, dass ein Deadlock vorliegt, wenn zwei Transaktionen zu lange brauchen und nicht beendet werden. Aber anders als Java kann der Anwendungsserver dann ein »Transaktions-Rollback« durchführen, mit dem sich das System wieder in den Status zurückversetzen lässt, in dem es vor Beginn der Transaktion (des atomaren Teils) war.

Java hat keinen Mechanismus zur Behandlung von Deadlocks. Es *merkt* noch nicht einmal, dass ein Deadlock aufgetreten ist. Es liegt also an Ihnen, sorgfältig zu programmieren. Wenn Sie viel Multithreading-Code schreiben werden, sollten Sie vielleicht »Java Threads« von Scott Oaks und Henry Wong (O'Reilly) lesen; dort finden Sie Designtipps zur Vermeidung von Deadlocks. Einer der häufigsten Tipps ist, auf die Reihenfolge zu achten, in der Ihre Threads gestartet werden.

**Für einen Deadlock genügen bereits zwei Objekte und zwei Threads.**




## Ein einfaches Deadlock-Szenario:



## Punkt für Punkt

- Die statische Methode Thread.sleep() zwingt einen Thread, den Zustand »laufend« aufzugeben und mindestens während der der sleep-Methode übergebenen Zeitdauer zu schlafen. Thread.sleep(200) lässt einen Thread 200 Millisekunden schlafen.
- Die Methode sleep() wirft eine geprüfte Exception (InterruptedException), daher müssen alle Aufrufe von sleep() mit einem try/catch-Block umgeben oder deklariert werden.
- Durch die Verwendung von sleep() können Sie sicherstellen, dass alle Threads eine Chance erhalten zu laufen. Allerdings ist nicht garantiert, dass sich ein Thread beim Erwachen am Ende der Reihe der Lauffähigen anstellt. Stattdessen könnte er sich beispielsweise gleich wieder nach vorn drängen. In den meisten Fällen können Sie Ihre Threads durch zeitlich zweckmäßig abgestimmte Aufrufe von sleep() dazu bringen, sich schön abzuwechseln.
- Sie können einem Thread mit der Methode setName() einen Namen geben. Alle Threads erhalten standardmäßig einen Namen, aber wenn Sie Ihre Threads explizit selbst benennen, können Sie ihre Spuren verfolgen, vor allem bei der Fehlersuche mit Hilfe von print-Anweisungen.
- Threads können gravierende Probleme verursachen, wenn zwei oder mehr Threads Zugriff auf das gleiche Objekt auf dem Heap haben.
- Der Zugriff von zwei oder mehr Threads auf das gleiche Objekt kann zu Datenverfälschung führen, wenn beispielsweise einer der beiden Threads den Zustand »laufend« verlässt, während er gerade dabei ist, den kritischen Zustand eines Objekts zu manipulieren.
- Um Ihre Objekte threadsicher zu machen, legen Sie fest, welche Anweisungen als *ein* atomarer Prozess behandelt werden sollten. Mit anderen Worten, legen Sie fest, welche Methoden vollständig ablaufen müssen, bevor irgendein anderer Thread Zutritt zur selben Methode desselben Objekts erhält.
- Modifizieren Sie eine Methodendeklaration mit dem Schlüsselwort **`synchronized`**, wenn Sie zwei Threads daran hindern wollen, diese Methode gleichzeitig zu benutzen.
- Jedes Objekt hat genau *ein* Schloss (einen Lock), und es gibt nur einen einzigen Schlüssel dafür. Meistens ist das Schloss für uns uninteressant; Schlösser kommen erst dann ins Spiel, wenn ein Objekt synchronisierte Methoden hat.
- Wenn ein Thread versucht, in eine synchronisierte Methode einzutreten, muss er sich den Schlüssel für das Objekt holen (den des Objekts, dessen Methode der Thread auszuführen versucht). Wenn der Schlüssel nicht verfügbar ist (weil ihn schon ein anderer Thread hat), geht der Thread in eine Art Warteraum, bis der Schlüssel verfügbar wird.
- Selbst wenn ein Objekt mehr als *eine* synchronisierte Methode hat, gibt es nicht mehr als nur *einen* Schlüssel. Solange sich irgendein Thread in einer synchronisierten Methode in diesem Objekt befindet, kann kein anderer Thread in irgendeine andere synchronisierte Methode desselben Objekts eintreten. Mit Hilfe dieser Einschränkung können Sie Ihre Daten schützen, indem Sie alle Methoden, die die Daten manipulieren, synchronisieren.

# Verbesserter EinfacherChatClient

Vor einer ganzen Weile, zu Beginn dieses Kapitels, wurde ein EinfacherChatClient von uns gebaut, der zwar ausgehende Nachrichten an den Server *senden*, aber nichts empfangen konnte. Erinnern Sie sich? So sind wir überhaupt erst in dieses ganze Thread-Thema geraten, weil wir eine Möglichkeit brauchten, zwei Dinge auf einmal zu tun: Nachrichten *an* den Server senden (durch Interaktion mit der Benutzeroberfläche) sowie gleichzeitig *vom* Server eingehende Nachrichten lesen und sie im scrollbaren Textbereich anzeigen.

Ja, auch dieses Kapitel ist irgendwann zu Ende. Aber noch nicht ...

```
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class EinfacherChatClient {

  JTextArea eingehend;
  JTextField ausgehend;
  BufferedReader reader;
  PrintWriter writer;
  Socket sock;

  public static void main(String[] args) {
    EinfacherChatClient client = new EinfacherChatClient();
    client.los();
  }

  public void los() {

    JFrame frame = new JFrame("Lächerlich einfacher Chat-Client");
    JPanel hauptPanel = new JPanel();
    eingehend = new JTextArea(15,20);
    eingehend.setLineWrap(true);
    eingehend.setWrapStyleWord(true);
    eingehend.setEditable(false);
    JScrollPane fScroller = new JScrollPane(eingehend);
    fScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
    fScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
    ausgehend = new JTextField(20);
    JButton sendenButton = new JButton("Senden");
    sendenButton.addActionListener(new SendenButtonListener());
    hauptPanel.add(fScroller);
    hauptPanel.add(ausgehend);
    hauptPanel.add(sendenButton);
    netzwerkEinrichten();

    Thread readerThread = new Thread(new EingehendReader());
    readerThread.start();

    frame.getContentPane().add(BorderLayout.CENTER, hauptPanel);
    frame.setSize(400,500);
    frame.setVisible(true);

  } // los schließen
```

Dies ist größtenteils GUI-Code, den Sie schon mal gesehen haben. Nichts Besonderes, abgesehen von dem hervorgehobenen Teil, in dem wir den neuen >>reader<<-Thread starten.

Wir starten einen neuen Thread und benutzen dabei eine neue innere Klasse als Runnable (Job) für den Thread. Der Job des Threads ist es, vom Socket-Strom des Servers zu lesen und alle eingehenden Nachrichten im scrollbaren Textbereich anzuzeigen.

```
  private void netzwerkEinrichten() {

    try {
      sock = new Socket("127.0.0.1", 5000);
      InputStreamReader streamReader = new InputStreamReader(sock.getInputStream());
      reader = new BufferedReader(streamReader);
      writer = new PrintWriter(sock.getOutputStream());
      System.out.println("Netzwerkverbindung steht");
    } catch(IOException ex) {
      ex.printStackTrace();
    }
  } // netzwerkEinrichten schließen
```

Wir verwenden den Socket, um an die Eingabe- und Ausgabeströme zu gelangen. Den Ausgabestrom benutzen wir bereits, um an den Server zu senden; jetzt benutzen wir auch den Eingabestrom, damit der neue >>reader<<-Thread Nachrichten vom Server erhalten kann.

```
  public class SendenButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      try {
        writer.println(ausgehend.getText());
        writer.flush();

      } catch(Exception ex) {
        ex.printStackTrace();
      }
      ausgehend.setText("");
      ausgehend.requestFocus();
    }
  }  // innere Klasse SendenButtonListener schließen
```

Nichts Neues. Wenn der Benutzer auf den Senden-Button klickt, sendet diese Methode den Inhalt des Textfelds an den Server.

```
  public class EingehendReader implements Runnable {
    public void run() {
      String nachricht;
      try {

        while ((nachricht = reader.readLine()) != null) {
          System.out.println("gelesen: " + nachricht);
          eingehend.append(nachricht + "\n");

        } // Ende der while-Schleife
      } catch(Exception ex) {ex.printStackTrace();}
    } // run schließen
  } // innere Klasse EingehendReader schließen
} // äußere Klasse schließen
```

Hier ist das, was der Thread tut!!!

Er bleibt in einer Schleife in der run()-Methode (solange das, was vom Server kommt, nicht null ist), liest zeilenweise aus dem Strom und setzt jede gelesene Zeile in den scrollbaren Textbereich (ergänzt durch einen Zeilenumbruch).

Code-Fertiggericht

# Der wirklich wirklich einfache Chat-Server

Sie können diesen Server-Code für beide Versionen des Chat-Clients verwenden. Hier gilt allerdings ausdrücklich jeder nur denkbare Haftungsausschluss. Um den Code auf das unbedingt Notwendige zu beschränken, haben wir eine Menge Teile herausgenommen, die man brauchen würde, um daraus einen echten Server zu machen. Mit anderen Worten, er funktioniert, aber es gibt mindestens hundert Arten, ihn kaputtzumachen. Falls Ihnen nach einer *wirklich guten* Spitzen-Sie-Ihren-Bleistift-Übung ist, wenn Sie das Buch durchhaben, dann blättern Sie hierher zurück und machen diesen Server-Code robuster.

Eine weitere Bleistift-Übung, die Sie gleich jetzt machen könnten, ist dies: Schreiben Sie die Anmerkungen zu diesem Code selbst daneben. Sie werden ihn wesentlich besser verstehen, wenn Sie sich selbst erarbeiten, was passiert, als wenn wir es Ihnen erklären. Andererseits – es ist Fertiggericht-Code, und Sie müssen eigentlich nicht alles verstehen. Er steht hier nur zur Unterstützung der beiden Chat-Client-Versionen.

**Um den Chat-Client auszuführen, brauchen Sie zwei Kommandozeilen-Fenster. Starten Sie als Erstes diesen Server in einem Kommandozeilen-Fenster, dann starten Sie den Client in einem zweiten.**

```
import java.io.*;
import java.net.*;
import java.util.*;

public class SehrEinfacherChatServer {

  ArrayList clientAusgabeStröme;

  public class ClientHandler implements Runnable {
    BufferedReader reader;
    Socket sock;

    public ClientHandler(Socket clientSocket) {
      try {
        sock = clientSocket;
        InputStreamReader isReader = new InputStreamReader(sock.getInputStream());
        reader = new BufferedReader(isReader);

      } catch(Exception ex) {ex.printStackTrace();}
    } // Konstruktor schließen

    public void run() {
      String nachricht;
      try {
        while ((nachricht = reader.readLine()) != null) {
          System.out.println("gelesen: " + nachricht);
          esAllenWeitersagen(nachricht);

        } // Ende der while-Schleife
      } catch(Exception ex) {ex.printStackTrace();}
    } // run schließen
  } // innere Klasse schließen
```

```
  public static void main (String[] args) {
    new SehrEinfacherChatServer().los();
  }

  public void los() {
    clientAusgabeStröme = new ArrayList();

    try {
      ServerSocket serverSock = new ServerSocket(5000);

      while(true) {
        Socket clientSocket = serverSock.accept();
        PrintWriter writer = new PrintWriter(clientSocket.getOutputStream());
        clientAusgabeStröme.add(writer);
        Thread t = new Thread(new ClientHandler(clientSocket));
        t.start();
        System.out.println("habe eine Verbindung");
      }
      // wenn wir hier angelangt sind, haben wir eine Verbindung

    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  public void esAllenWeitersagen(String nachricht) {

    Iterator it = clientAusgabeStröme.iterator();
    while(it.hasNext()) {
      try {
        PrintWriter writer = (PrintWriter) it.next();
        writer.println(nachricht);
        writer.flush();
      } catch(Exception ex) {
        ex.printStackTrace();
      }

    } // Ende der while-Schleife

  } // esAllenWeitersagen schließen
} // Klasse schließen
```

## Es gibt keine Dummen Fragen

**F: Was ist mit dem Schutz des Zustands von statischen Variablen? Wenn Sie statische Methoden haben, die den Zustand einer statischen Variablen verändern, können Sie dann auch Synchronisierung einsetzen?**

A: Ja! Erinnern Sie sich, dass statische Methoden auf der Klasse ausgeführt werden, nicht auf einer einzelnen Instanz der Klasse. Sie fragen sich vielleicht, das Schloss welchen Objekts Sie bei einer statischen Methode verwenden würden? Schließlich könnte es ja sein, dass keinerlei Instanz der Klasse existiert. Glücklicherweise hat genau wie jedes *Objekt* auch jede geladene *Klasse* ihr eigenes Schloss. Wenn Sie also auf Ihrem Heap drei Hund-Objekte haben, haben Sie insgesamt vier Hund-Schlösser. Drei davon gehören zu den drei Hund-Instanzen, und eins gehört zur Klasse Hund selbst. Wenn Sie eine statische Methode synchronisieren, benutzt Java das Schloss, das zur Klasse gehört. Synchronisieren Sie also zwei statische Methoden in ein und derselben Klasse, braucht ein Thread das Klassenschloss für beide Methoden, um in eine der beiden Methoden eintreten zu können.

**F: Was sind Thread-Prioritäten? Ich habe gehört, das sei eine Möglichkeit, den Scheduler zu steuern.**

A: Thread-Prioritäten *können* Ihnen vielleicht helfen, den Scheduler zu beeinflussen, aber auch sie bieten keine Garantie. Thread-Prioritäten sind numerische Werte, die dem Scheduler sagen (falls es ihn überhaupt interessiert), wie wichtig ein Thread Ihnen ist. Im Allgemeinen nimmt der Scheduler einen Thread mit niedrigerer Priorität aus dem Zustand »laufend« heraus, wenn ein Thread mit höherer Priorität plötzlich »lauffähig« wird. Aber ... noch einmal (sprechen Sie es laut mit): »Es gibt keine Garantie.« Wir empfehlen Ihnen, Prioritäten nur zur Verbesserung der Leistungsfähigkeit Ihres Programms einzusetzen, aber verlassen Sie sich *auf gar keinen Fall* auf sie, wenn es um die korrekte Ausführung des Programms geht.

**F: Warum synchronisiert man nicht einfach alle Getter und Setter in der Klasse mit den schützenswerten Daten? Warum hätten wir z.B. nicht einfach die Methoden getKontostand() und abbuchen() in der Klasse BankKonto synchronisieren können statt die Methode machAbhebung() in der Runnable-Klasse?**

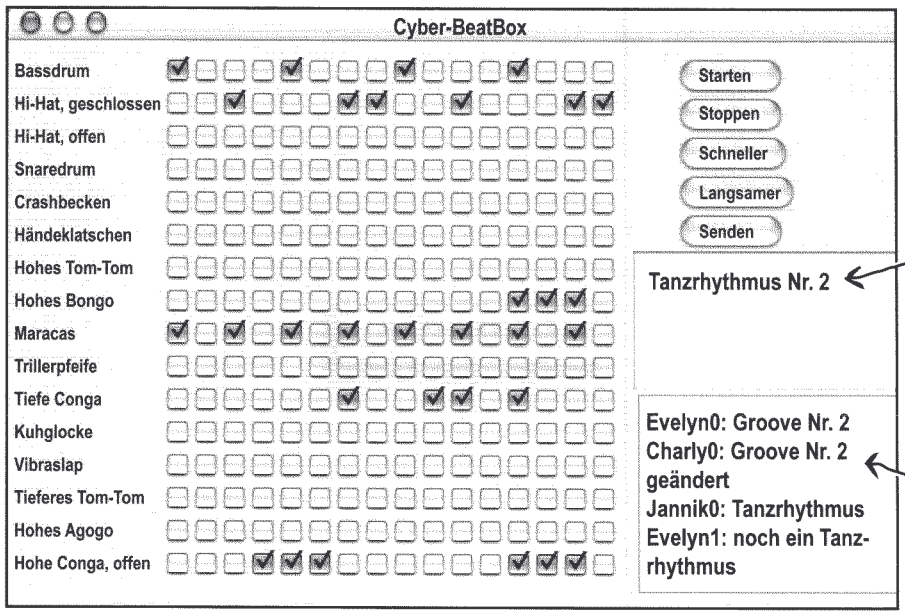
A: In der Tat hätten wir diese Methoden eigentlich synchronisieren sollen, um andere Threads daran zu hindern, auf anderem Weg auf diese Methoden zuzugreifen. Wir haben uns jedoch nicht die Mühe gemacht, weil es in unserem Beispiel keinen anderen Code für den Zugriff auf das Konto gab.

Aber mit dem Synchronisieren der Getter und Setter (oder in diesem Fall der Methoden getKontostand() und abbuchen()) allein ist es nicht getan. Erinnern Sie sich, beim Synchronisieren geht es darum, dass ein bestimmter Codeabschnitt ATOMAR arbeiten soll. Anders ausgedrückt, es sind nicht einfach die einzelnen Methoden, die uns interessieren, sondern es sind Methoden, ***die zu ihrer vollständigen Ausführung mehrere Schritte benötigen!*** Denken Sie darüber nach. Wenn wir die Methode machAbhebung() nicht synchronisiert hätten, hätte Rainer den Kontostand geprüft (durch Aufruf der synchronisierten Methode getKontostand()) und dann die Methode sofort wieder verlassen und den Schlüssel zurückgegeben.

Natürlich würde er sich den Schlüssel zurückholen, sobald er aufgewacht ist, um die synchronisierte Methode abbuchen() aufzurufen, aber dann haben wir noch immer das gleiche Problem, das wir vor der Synchronisierung hatten! Rainer kann den Kontostand prüfen, einschlafen, und dann kann Monika kommen und ebenfalls den Kontostand abfragen, bevor Rainer eine Chance hat, aufzuwachen und seine Abhebung zu Ende durchzuführen.

Die Synchronisierung aller Zugriffsmethoden ist also wahrscheinlich eine gute Idee, um andere Threads vom Zugriff abzuhalten, aber trotzdem müssen Sie noch die Methoden synchronisieren, in denen Anweisungen vorkommen, die als atomare Einheit auszuführen sind.

# CodeKüche



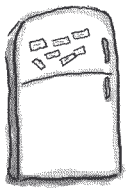

Wenn Sie auf >>Senden<< klicken, wird Ihre Nachricht zusammen mit Ihrem aktuellen Beat-Pattern an die anderen Schlagzeuger geschickt.

Eingehende Nachrichten von Teilnehmern. Klicken Sie auf eine davon, um das Pattern zu laden, das dazugehört, und dann klicken Sie auf >>Start<<, um es abzuspielen.

**Dies ist die letzte Version der BeatBox!**

**Sie verbindet sich mit einem einfachen MusikServer, damit Sie Beat-Patterns mit anderen Clients austauschen können.**

**Der vollständige Code ist wirklich lang, deshalb ist er nicht hier, sondern in Anhang A abgedruckt.**



# Code-Magneten

Jemand hat ein funktionsfähiges Java-Programm in kleinen Schnipseln an den Kühlschrank geheftet. Können Sie alle Codeschnipsel wieder zu einem funktionierenden Java-Programm zusammensetzen, das die unten abgedruckte Ausgabe erzeugt? Ein paar geschweifte Klammern sind auf den Boden gefallen, und weil sie so klein sind, hat sie niemand aufgehoben – Sie können also so viele davon einsetzen, wie Sie benötigen!

```
public class TestThreads {
```

```
class Akkum {
```

```
class ThreadEins
```

```
class ThreadZwei
```

```
Datei Bearbeiten Fenster Hilfe SpendenAktion
% java TestThreads
eins: 98098
zwei: 98099
```

**Bonusfrage:** Was glauben Sie – warum haben wir in der Klasse Akkum gerade diese Modifizierer gewählt?

# Code-Magneten (Fortsetzung)

# Lösungen zu den Übungen

```
public class TestThreads {
  public static void main(String [] args) {
    ThreadEins t1 = new ThreadEins();
    ThreadZwei t2 = new ThreadZwei();
    Thread eins = new Thread(t1);
    Thread zwei = new Thread(t2);
    eins.start();
    zwei.start();
  }
}

class Akkum {
  private static Akkum a = new Akkum();
  private int anzahl = 0;

  private Akkum() { }

  public static Akkum getAkkum() {
    return a;
  }

  public void aktualisiereAnzahl(int bei-
trag) {
    anzahl += beitrag;
  }

  public int getAnzahl() {
    return anzahl;
  }
}

class ThreadEins implements Runnable {
  Akkum a = Akkum.getAkkum();
  public void run() {
    for(int x=0; x < 98; x++) {
      a.aktualisiereAnzahl(1000);
      try {
        Thread.sleep(50);
      } catch(InterruptedException ex) { }
    }
    System.out.println("eins: " +
a.getAnzahl());
  }
}
```

eine statische Instanz der Klasse Akkum erzeugen

ein privater Konstruktor

Threads aus zwei verschiedenen Klassen aktualisieren das gleiche Objekt in einer dritten Klasse, da beide Threads auf die gleiche Instanz von Akkum zugreifen. Die Codezeile

```
private static Akkum a = new Akkum();
```

erzeugt eine statische Instanz von Akkum (erinnern Sie sich, statisch heißt »eine pro Klasse«), und der private Konstruktor in Akkum bedeutet, dass niemand anderes ein Akkum-Objekt erzeugen kann. Diese beiden Techniken (privater Konstruktor und statische Getter-Methode) bilden bei gemeinsamer Anwendung etwas, das man als Singleton bezeichnet - ein OO-Muster, mit dem man die Anzahl der Instanzen beschränkt, die es von einem Objekt innerhalb einer Anwendung geben darf. (In der Regel gibt es nur eine einzige Instanz eines Singletons - daher der Name, aber Sie können mit dem Muster die Instanzerzeugung auch in anderer Weise begrenzen, ganz wie Sie möchten.)

```
class ThreadZwei implements Runnable {
  Akkum a = Akkum.getAkkum();
  public void run() {
    for(int x=0; x < 99; x++) {
      a.aktualisiereAnzahl(1);
      try {
        Thread.sleep(50);
      } catch(InterruptedException ex) { }
    }
    System.out.println("zwei: " +
a.getAnzahl());
  }
}
```

## Ein Luftschleusenalbtraum

Als Sarah zur Design-Review-Besprechung des Bord-Entwicklerteams kam, warf sie durch das Tor einen Blick auf den Sonnenaufgang über dem Indischen Ozean. Der Konferenzraum auf dem Schiff war zwar erdrückend eng, doch der Anblick der wachsenden blauweißen Sichel, die die Nacht auf dem Planeten verdrängte, erfüllte sie mit einem Gefühl von Ehrfurcht und Dankbarkeit.

Kurzgeschichte

Bei der Besprechung ging es um die Steuerungssysteme für die Luftschleusen des Orbiters. Da der letzte Bauabschnitt fast abgeschlossen war, plante man eine drastische Zunahme der Spaziergänge im All, und in den Schleusen des Schiffs herrschte in beiden Richtungen reger Betrieb. »Guten Morgen, Sarah«, sagte Tim, »du kommst genau zur rechten Zeit, wir fangen gerade mit der gründlichen Design-Überprüfung an.«

»Wie ihr alle wisst«, fuhr Tim fort, »ist jede Luftschleuse mit raumgehärteten GUI-Terminals ausgerüstet, sowohl innen als auch außen. Immer wenn Raumspaziergänger in den Orbiter zurückkehren oder ihn verlassen, setzen sie die Abläufe in der Luftschleuse mit Hilfe dieser Terminals in Gang.« Sarah nickte. »Tim, kannst du uns sagen, wie die Methodenabfolge beim Verlassen und Wiederbetreten des Schiffs ist?« Tim erhob sich und schwebte zur Tafel: »Hier erst einmal der Pseudocode für die Sequenz zum Verlassen ...«, und er schrieb ihn rasch an die Tafel:

```
orbiterVerlassenLuftschleusenSequenz() }
   torStatusVerifizieren();
   schleuseUnterDruckSetzen();
   innereLukeÖffnen();
   bestätigenSchleuseBesetzt();
   innereLukeSchließen();
   schleuseDruckentlasten();
   äußereLukeÖffnen();
   bestätigenSchleuseLeer();
   äußereLukeSchließen();
}
```

»Um sicherzustellen, dass die Sequenz nicht unterbrochen wird, haben wir alle Methoden, die von der Methode orbiterVerlassenLuftschleusenSequenz() aufgerufen werden, synchronisiert«, erklärte Tim. »Wir möchten natürlich vermeiden, dass zurückkehrende Raumspaziergänger unbeabsichtigt einen Kollegen mit heruntergelassenen Raumanzughosen überraschen!«

Alle lachten, während Tim die Tafel abwischte, aber Sarah hatte das Gefühl, dass irgendetwas nicht stimmte. Der Groschen fiel, als Tim begann, den Pseudocode der Sequenz für das Wiederbetreten an die Tafel zu schreiben. »Warte mal, Tim!« rief sie. »Ich glaube, wir haben da einen dicken Fehler im Design der Verlassen-Sequenz! Lass uns das noch mal ansehen, es könnte kritisch sein!«

*Warum unterbrach Sarah die Besprechung? Welchen Verdacht hatte sie?*

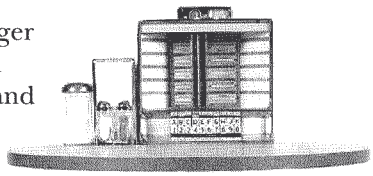
## Was hatte Sarah erkannt?

Sarah war klar geworden: Wenn die gesamte Verlassen-Sequenz ohne Unterbrechung ablaufen sollte, musste auch die Methode `orbiterVerlassenLuftschleusenSequenz()` synchronisiert werden. In der jetzigen Fassung des Designs hätte ein zurückkehrender Raumspaziergänger die Sequenz unterbrechen können. Es war zwar keine Unterbrechung mitten in einem Aufruf einer Lowlevel-Methode möglich, aber *zwischen* diesen Aufrufen konnte die Sequenz sehr wohl unterbrochen werden. Sarah wusste, dass die gesamte Sequenz besser als *eine* atomare Einheit ausgeführt werden sollte. Wenn die Methode `orbiterVerlassenLuftschleusenSequenz()` synchronisiert würde, könnte sie dagegen an keiner Stelle mehr unterbrochen werden.

# Datenstrukturen

**Sortieren ist in Java ein Klacks.** Sie haben alle Werkzeuge zur Verfügung, um Daten zu sammeln und damit zu arbeiten, ohne Ihre eigenen Sortieralgorithmen schreiben zu müssen (falls Sie nicht gerade in einem Informatik-Anfängerseminar sitzen; dort werden Sie – glauben Sie uns – *ganz sicher* Sortiercode schreiben, während wir anderen einfach eine Methode in der Java-API aufrufen). Das Collections-Framework von Java enthält eine Datenstruktur für praktisch alles, was Sie jemals tun müssen. Sie wollen eine Liste, zu der Sie leicht etwas hinzufügen können? Sie wollen etwas mit Hilfe seines Namens suchen? Oder Sie brauchen eine Liste, die automatisch alles entfernt, was doppelt vorhanden ist? Sie möchten Ihre Kollegen danach sortieren, wie oft sie Ihnen in den Rücken gefallen sind? Ihre Haustiere nach der Anzahl der gelernten Kunststücke sortieren? Es ist alles da ...

# Die Beliebtheit eines Songs in Ihrer Jukebox verfolgen



Herzlichen Glückwunsch zu Ihrem neuen Job! Sie sind Manager für das automatisierte Jukebox-System in Jupps Restaurant. In der Jukebox selbst steckt kein Java, aber jedes Mal, wenn jemand einen Song spielt, werden die Songdaten an eine einfache Textdatei angehängt.

Ihre Aufgabe ist es, die Daten zu verwalten und damit die Beliebtheit der Songs zu verfolgen, Berichte zu schreiben und die Playlisten zu bearbeiten. Sie schreiben nicht das gesamte Programm selbst, ein paar der anderen Softwareentwickler/Kellner arbeiten ebenfalls daran mit. Aber Sie sind dafür verantwortlich, dass die Daten innerhalb des Java-Programms verwaltet und sortiert werden. Und da Jupp etwas gegen Datenbanken hat, bleibt das streng auf eine speicherresidente Datensammlung beschränkt. Sie bekommen lediglich die Datei, die von der Jukebox laufend fortgeschrieben wird. Es ist Ihr Job, die Daten dort herauszuholen.

Sie haben schon herausbekommen, wie man die Datei liest und parst, und bisher haben Sie die Daten in einer ArrayList gespeichert.

**SongList.txt**

```
Pink Moon/Nick Drake
Somersault/Zero 7
Shiva Moon/Prem Joshua
Circles/BT
Deep Channel/Afro Celts
Passenger/Headmix
Listen/Tahiti 80
```

Dies ist die Datei, die der Jukebox-Automat schreibt. Ihr Code muss die Datei lesen und dann die Song-Daten verarbeiten.

## Erste Herausforderung
## Die Songs alphabetisch sortieren

Sie haben eine Liste mit Songs in einer Datei, wobei jede Zeile einen Song repräsentiert und Titel und Künstler durch einen Schrägstrich getrennt sind. Es sollte daher einfach sein, die Zeile zu parsen und alle Songs in eine ArrayList zu setzen.

Ihr Boss interessiert sich nur für die Titel der Songs; vorläufig können Sie daher einfach eine Liste machen, die nur die Songtitel enthält.

Aber wie Sie sehen, ist die Liste nicht alphabetisch sortiert ... was kann man da tun?

Sie wissen, dass die Elemente bei einer ArrayList die Reihenfolge behalten, in der sie in die Liste eingefügt wurden. Wenn man sie also in eine ArrayList steckt, wird diese nicht für eine alphabetische Sortierung sorgen, außer ... wenn es in der Klasse ArrayList vielleicht eine Methode sort() gibt!

# Bis jetzt hatten Sie das hier, ohne Sortierung:

```
import java.util.*;
import java.io.*;

public class Jukebox1 {

  ArrayList<String> songList = new ArrayList<String>();

  public static void main(String[] args) {
    new Jukebox1().los();
  }

  public void los() {
    getSongs();
    System.out.println(songList);
  }

  void getSongs() {
    try {
      File datei = new File("SongList.txt");
      BufferedReader reader = new BufferedReader(new FileReader(datei));
      String zeile = null;
      while ((zeile= reader.readLine()) != null) {
        addSong(zeile);
      }

    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  void addSong(String zuParsendeZeile) {
    String[] tokens = zuParsendeZeile.split("/");
    songList.add(tokens[0]);
  }

}
```

Wir speichern die Songtitel einer ArrayList von Strings.

Die Methode, die das Laden der Datei startet und dann den Inhalt der ArrayList songList ausgibt.

Hier nichts Besonderes ... lesen Sie einfach die Datei und rufen Sie auf jeder Zeile die Methode addSong() auf.

Die addSong-Methode funktioniert genau wie die bei den Quizkarten im E/A-Kapitel – Sie teilen die Zeile (in der sowohl Titel als auch Künstler stehen) mit Hilfe der Methode split() in zwei Stücke (Tokens).

Wir wollen nur den Songtitel und fügen daher nur das erste Token der songList (der ArrayList) hinzu.

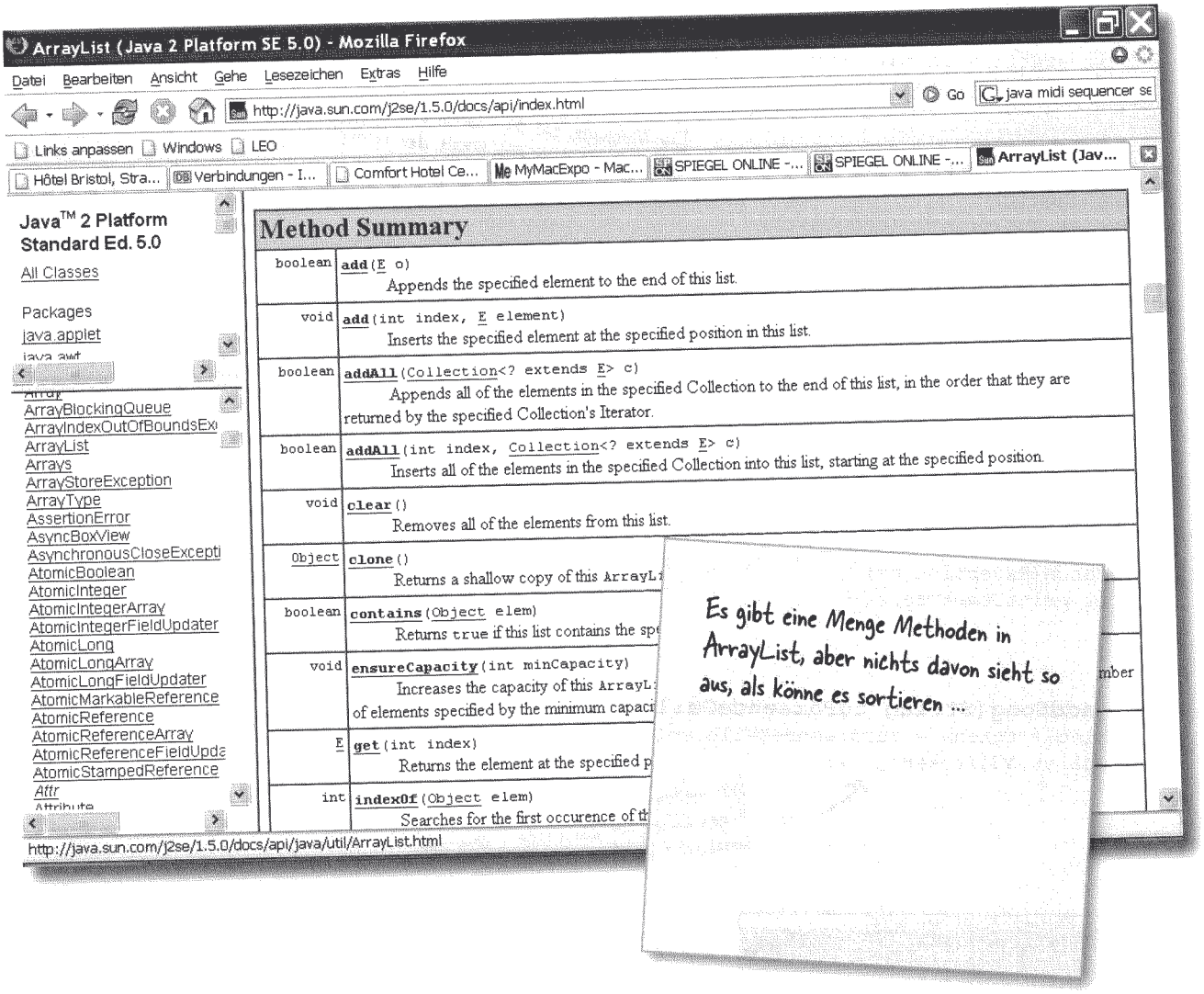
```
Datei Bearbeiten Fenster Hilfe Tanzen
% java Jukebox1
[Pink Moon, Somersault,
Shiva Moon, Circles,
Deep Channel, Passenger,
Listen]
```

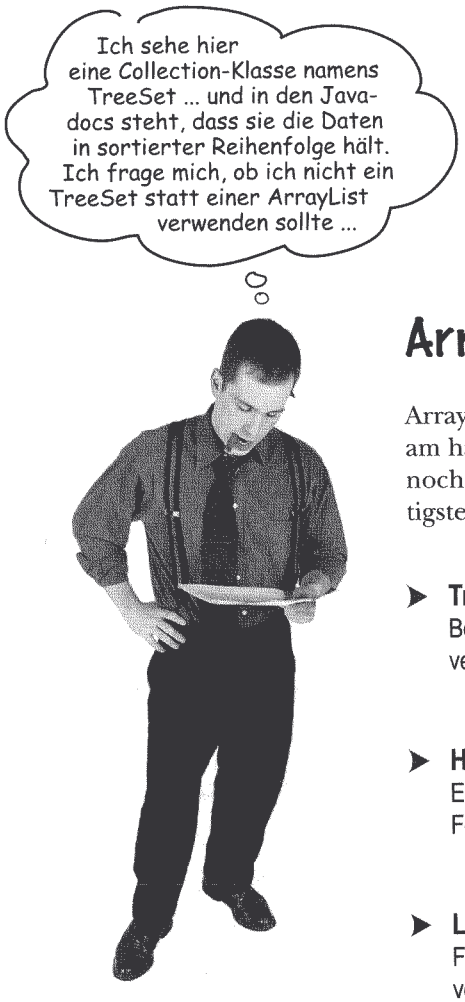
Die Songs aus der Songliste werden in der Reihenfolge ausgegeben, in der sie zur ArrayList hinzufügt wurden (das ist die gleiche Reihenfolge, in der sie in der ursprünglichen Textdatei stehen).

Und das ist eindeutig NICHT alphabetisch!

# Aber die Klasse ArrayList hat gar keine Methode sort()!

Sie können unter ArrayList nachschlagen, aber dort scheint es keine Methode zu geben, die etwas mit Sortieren zu tun hat. In der Vererbungshierarchie weiter nach oben zu gehen hat uns ebenfalls nicht geholfen – ***ganz eindeutig können Sie auf einer ArrayList keine Sortiermethode aufrufen.***






# ArrayList ist nicht die einzige Collection

ArrayList ist zwar die Collection, die Sie wahrscheinlich am häufigsten einsetzen werden, doch gibt es daneben noch weitere für besondere Gelegenheiten. Zu den wichtigsten Collection-Klassen gehören:

Keine Sorge, Sie brauchen diese anderen hier nicht sofort auswendig zu lernen. Wir gehen etwas weiter unten noch genauer darauf ein.

- **TreeSet**
  Bewahrt die Elemente in sortierter Reihenfolge auf und verhindert ein doppeltes Vorkommen.

- **HashMap**
  Ermöglicht die Speicherung und den Zugriff auf Elemente in Form von Name/Wert-Paaren.

- **LinkedList**
  Für eine schnellere Verarbeitung beim Einfügen oder Löschen von Elementen in der Mitte einer Collection. (In der Praxis werden Sie aber auch hierfür in der Regel eine ArrayList einsetzen.)

- **HashSet**
  Verhindert doppelte Vorkommen von Elementen in der Collection und kann ein gegebenes Element in der Collection schnell finden.

- **LinkedHashMap**
  Wie eine normale HashMap, kann sich jedoch die Reihenfolge merken, in der Elemente (Name/Wert-Paar) eingefügt wurden, oder so konfiguriert werden, dass sie sich die Reihenfolge merkt, in der zuletzt auf die Elemente zugegriffen wurde.

# Sie *könnten* ein TreeSet verwenden ... oder die Methode Collections.sort()

Wenn Sie alle Strings (die Songtitel) in ein **TreeSet** setzen würden statt in eine ArrayList, würden die Strings automatisch an der richtigen Stelle landen, alphabetisch sortiert. Beim Ausdrucken der Liste kämen die Elemente dann immer in alphabetischer Reihenfolge heraus.

Und das ist prima, wenn Sie ein *Set* brauchen (darüber sprechen wir gleich noch) oder wenn Sie wissen, dass die Liste *immer* alphabetisch sortiert bleiben muss.

Muss Ihre Liste dagegen nicht in alphabetischer Reihenfolge bleiben, ist TreeSet möglicherweise mit mehr Aufwand verbunden als nötig – ***jedes Mal, wenn Sie etwas in ein TreeSet einfügen, muss sich das TreeSet die Zeit nehmen, um herauszufinden, wo das neue Element in den Baum einzusortieren ist.*** Bei einer ArrayList kann das Einfügen dagegen blitzschnell gehen, da das neue Element einfach am Ende hineingesetzt wird.

**java.util.Collections**

public static void **copy**(List destination, List source)

public static List **emptyList**()

public static void **fill**(List listToFill, Object objToFillItWith)

public static int **frequency**(Collection c, Object o)

public static void **reverse**(List list)

public static void **rotate**(List list, int distance)

public static void **shuffle**(List list)

public static void **sort**(List list)

public static boole... List list, Object oldVal, Object newVal)

// viele weitere M...

Hmmm ... es gibt tatsächlich eine Methode sort() in der Klasse Collections. Sie erwartet eine List, und da ArrayList das Interface List implementiert, gilt: ArrayList IST-EINE List. Dank Polymorphie können Sie eine ArrayList an eine Methode übergeben, die als Argument eine List deklariert.

**F: Aber man kann doch etwas bei einem bestimmten Index zu einer ArrayList hinzufügen, nicht nur am Ende – es gibt nämlich eine überladene Methode add(), die neben dem hinzuzufügenden Element ein int erwartet. Wäre das auch langsamer als das Einfügen am Ende?**

A: Ja, wenn man in einer ArrayList etwas *woanders* als am Ende einfügt, dauert es länger. Mit der überladenen Methode add(index,element) geht es also nicht so schnell wie beim Aufruf der Methode add(element) – die das hinzugefügte Element ans Ende setzt. Es wird jedoch bei Ihrer Arbeit mit ArrayLists nur selten erforderlich sein, dass Sie etwas bei einem bestimmten Index einfügen.

**F: Wie ich sehe, gibt es eine Klasse LinkedList; wäre *die* nicht besser geeignet, um irgendwo in der Mitte etwas einzufügen? Zumindest wenn ich an meinen Kurs »Datenstrukturen« während des Studiums zurückdenke ...**

A: Ja, das stimmt. Die LinkedList *kann* schneller sein, wenn Sie etwas in der Mitte einfügen oder entfernen, aber bei den meisten Programmen ist der Unterschied zwischen LinkedList und ArrayList beim Einfügen in der Mitte normalerweise nicht so groß, dass man sich darüber Gedanken machen müsste – es sei denn, es geht um *sehr viele* Elemente. Wir sehen uns LinkedList gleich noch näher an.

Anmerkung: Dies ist NICHT die echte API der Klasse Collections; wir haben sie hier vereinfacht und die Informationen über generische Typen weggelassen (das kommt ein paar Seiten weiter unten).

# Einbau von Collections.sort() in den Jukebox-Code

Die Methode Collections.sort() sortiert eine Liste von Strings alphabetisch.

```
import java.util.*;
import java.io.*;

public class Jukebox1 {

  ArrayList<String> songList = new ArrayList<String>();

  public static void main(String[] args) {
    new Jukebox1().los();
  }

  public void los() {
    getSongs();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);
  }

  void getSongs() {
    try {
      File datei = new File("SongList.txt");
      BufferedReader reader = new BufferedReader(new FileReader(datei));
      String zeile = null;
      while ((zeile= reader.readLine()) != null) {
        addSong(zeile);
      }

    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  void addSong(String zuParsendeZeile) {
    String[] tokens = zuParsendeZeile.split("/");
    songList.add(tokens[0]);
  }
}
```

Rufen Sie die statische Collections-Methode sort() auf und geben Sie dann die Liste erneut aus. Die zweite Ausgabe erfolgt in alphabetischer Reihenfolge!

```
Datei Bearbeiten Fenster Hilfe Chillen
%java Jukebox1

[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]
```

Vor dem Aufruf von sort().

Nach dem Aufruf von sort().

# Aber nun brauchen Sie Song-Objekte statt einfacher Strings

Jetzt möchte Ihr Boss richtige Instanzen einer Klasse Song in der Liste haben, nicht bloß Strings, damit zu jedem Song mehr Daten gehören können. Unser neuer Jukebox-Automat liefert mehr Informationen, daher wird die Datei jetzt *vier* Bruchstücke (Tokens) enthalten statt nur zwei.

Die Klasse Song ist wirklich einfach, interessant an ihr ist nur die überschriebene toString()-Methode. Wie Sie wissen, ist die Methode toString() in der Klasse Object definiert, und daher erbt jede Klasse in Java diese Methode. Und da toString() auf einem Objekt aufgerufen wird, wenn es ausgegeben wird (mit System.out.println(einObjekt)), sollten Sie diese Methode überschreiben, damit etwas ausgegeben wird, das besser lesbar ist als die standardmäßig vorgegebene eindeutige Kennung (unique identifier) für dieses Objekt. Beim Ausgeben einer Liste wird auf jedem Objekt die Methode toString() aufgerufen.

**SongListPlus.txt**

```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
```

Die neue Songdatei enthält vier Attribute statt nur zwei. Und diese sollen ALLE in unserer Liste enthalten sein, deshalb brauchen wir eine Klasse Song mit Instanzvariablen für alle vier Song-Attribute.

```
class Song {
  String titel;
  String künstler;
  String bewertung;
  String bpm;

  Song(String t, String k, String bw, String b) {
    titel = t;
    künstler = k;
    bewertung = bw;
    bpm = b;
  }

  public String getTitel() {
    return titel;
  }

  public String getKünstler() {
    return künstler;
  }

  public String getBewertung() {
    return bewertung;
  }

  public String getBpm() {
    return bpm;
  }

  public String toString() {
    return titel;
  }
}
```

Vier Instanzvariablen für die vier Song-Attribute in der Datei. (Die Variable bpm steht für die »beats per minute«, also die Geschwindigkeit des Songs.)

Die Variablen werden alle im Konstruktor gesetzt, wenn der neue Song erzeugt wird.

Die Getter-Methoden für die vier Attribute.

Wir überschreiben toString(), denn bei einem System.out.println(einSongObjekt) wollen wir den Titel sehen. Und bei einem System.out.println(eineListeVonSongs) wird die toString()-Methode von JEDEM Element der Liste aufgerufen.

# Umstellung der JukeBox von Strings auf Songs

Ihr Code verändert sich nur ein kleines bisschen – der Datei-E/A-Code ist der gleiche, das Parsen bleibt ebenfalls gleich (String.split()), abgesehen davon, dass wir dieses Mal *vier* Tokens pro Song/Zeile haben, die alle für die Erzeugung eines neuen Song-Objekts verwendet werden. Und natürlich ist die ArrayList vom Typ <Song> statt <String>.

```
import java.util.*;
import java.io.*;

public class Jukebox3 {

  ArrayList<Song> songList = new ArrayList<Song>();
  public static void main(String[] args) {
    new Jukebox3().los();
  }
  public void los() {
    getSongs();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);
  }
  void getSongs() {
    try {
      File datei = new File("SongListPlus.txt");
      BufferedReader reader = new BufferedReader(new FileReader(datei));
      String zeile = null;
      while ((zeile= reader.readLine()) != null) {
        addSong(zeile);
      }
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  void addSong(String zuParsendeZeile) {
    String[] tokens = zuParsendeZeile.split("/");

    Song nächsterSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
    songList.add(nächsterSong);
  }
}
```

Stellen Sie die ArrayList von Strings auf Song-Objekte um.

Erzeugen Sie mit den vier Tokens (den vier Informationsbruchstücken in der jeweiligen Song-Datei-Zeile) ein neues Song-Objekt und fügen Sie dieses anschließend der Liste hinzu.

# Es lässt sich nicht kompilieren!

Irgendwas stimmt nicht ... in der Klasse Collections steht doch eindeutig, dass es eine sort-Methode gibt, der eine List übergeben wird.

Und ArrayList IST-EINe List, weil ArrayList das Interface List implementiert ... es *sollte* also funktionieren.

***Tut es aber nicht!***

Der Compiler sagt, er kann keine Sortiermethode finden, der man eine ArrayList<Song> übergeben kann. Mag er vielleicht keine ArrayList mit Song-Objekten? Gegen eine ArrayListy<String> hatte er nichts; worin liegt also der wesentliche Unterschied zwischen Song und String? Der Unterschied, der zum Scheitern des Compilers führt?

```
Datei Bearbeiten Fenster Hilfe Bums
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol  : method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
                Collections.sort(songList);
                           ^
1 error
```

Und wahrscheinlich haben Sie sich auch schon selbst gefragt: »Wo*nach* soll denn eigentlich sortiert werden?« Woher soll die Sortiermethode überhaupt wissen, was einen Song »größer« oder »kleiner« macht als einen anderen Song? Wenn Sie möchten, dass die Songs nach dem Song-*Titel* sortiert werden, müssen Sie der Sortiermethode ganz offensichtlich irgendwie sagen, dass sie den Titel verwenden soll und nicht beispielsweise die Schläge pro Minute.

Darauf gehen wir ein paar Seiten später noch ein, aber erst einmal wollen wir herausfinden, warum uns der Compiler gar nicht erst erlaubt, der sort()-Methode eine Song-ArrayList zu übergeben.

## Die Deklaration der Methode sort()

Collections (Java 2 Platform SE 5.0)

file:///Users/kathy/Public/docs/api/index.html

Google

**Method Detail**

**sort**

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the *natural ordering* of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

Nach den API-Docs (wenn Sie bei der Klasse java.util.Collections nachsehen und dort zur Methode sort() scrollen) sieht es so aus, als wäre die Deklaration von sort() irgendwie ... *komisch.* Oder zumindest anders als alles, was wir bisher gesehen haben.

Das liegt daran, dass die Methode sort() (neben anderen Dingen im Collection-Framework von Java) umfassenden Gebrauch von *Generics** macht. Immer wenn Sie etwas in spitzen Klammern im Java-Quellcode oder in der Dokumentation sehen, heißt das, es geht um Generics – ein Sprachmittel, das mit Java 5.0 hinzugekommen ist. Es sieht also so aus, als müssten wir erst lernen, wie man die Dokumentation interpretiert, bevor wir herausfinden können, warum wir String-Objekte in einer ArrayList sortieren konnten, Song-Objekte dagegen nicht.

* Das englische Adjektiv *generic* heißt so viel wie »allgemein gehalten«. In unserem Zusammenhang bedeutet es, dass eine allgemein gehaltene Datenstruktur (die Collection) mit Datentypen parametrisiert werden kann. Das heißt, der Collection – dem »Sammelbehälter« – wird ein bestimmter Typ quasi wie ein Etikett aufgeklebt.

# Höhere Typsicherheit durch Generics

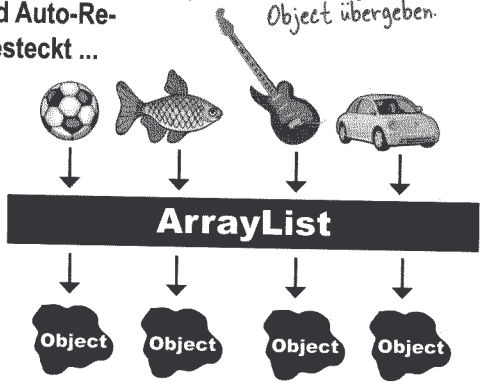
Eins können wir direkt sagen: *Wenn Sie Code schreiben, der etwas mit Generics zu tun hat, wird das praktisch immer in Verbindung mit Collections geschehen.* Es gibt zwar noch andere Anwendungsmöglichkeiten für Generics, aber das Wichtigste an ihnen ist, dass Sie damit typsichere Collections schreiben können. Anders ausgedrückt: Code, durch den der Compiler Sie daran hindern kann, einen Hund in eine Liste von Enten zu stecken.

Bevor es Generics gab (d.h. vor Java 5.0), scherte sich der Compiler nicht darum, was in eine Collection gesteckt wurde, denn alle Collection-Implementierungen waren so deklariert, dass sie den Typ Object enthielten. Man konnte einfach *alles* in eine ArrayList stecken; es war so, als wären alle ArrayLists als ArrayList<Object> deklariert.

**OHNE Generics**

**Objekte werden als Fußball-, Fisch-, Gitarre- und Auto-Referenzen HINEINgesteckt ...**

Bevor es Generics gab, hatte man keine Möglichkeit, den Typ einer ArrayList zu deklarieren, daher wurden seiner Methode add() Argumente vom Typ Object übergeben.



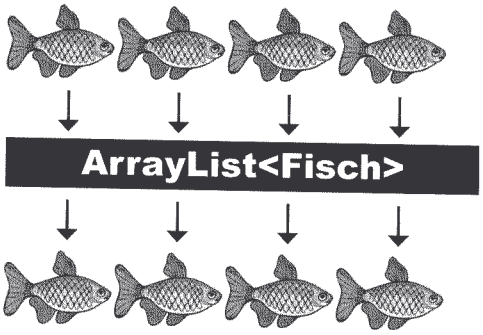

**... und kommen als Referenz vom Typ Object wieder HERAUS.**

Mit Generics können Sie typsichere Collections erzeugen, wodurch sich noch mehr Probleme schon zur Kompilierzeit abfangen lassen statt erst zur Laufzeit.

Ohne Generics durften Sie mit Genehmigung des Compilers einen Kürbis in eine ArrayList stecken, die eigentlich nur Objekte vom Typ Katze enthalten sollte.

**MIT Generics**

**Objekte werden ausschließlich als Referenzen auf Fisch-Objekte HINEINgesteckt ...**




Mit Generics können Sie jetzt nur Fisch-Objekte in die ArrayList<Fisch> stecken, daher kommen die Objekte auch als Fisch-Referenzen wieder heraus. Sie brauchen sich keine Sorgen zu machen, dass irgendjemand einen Volkswagen dort hineinsteckt oder dass das, was Sie herausholen, vielleicht nicht auf eine Fisch-Referenz gecastet werden kann.

**... und kommen auch als Referenz vom Typ Fisch wieder HERAUS.**

# Einführungskurs Generics

Es gibt Dutzende von Dingen, die Sie über Generics lernen könnten, aber für die meisten Programmierer sind eigentlich nur drei davon wichtig:

### 1 Erzeugung von Instanzen »generifizierter« *Klassen* (z.B. ArrayList)

Wenn Sie eine ArrayList erzeugen, müssen Sie ihr mitteilen, welchen Typ von Objekten Sie in der Liste zulassen – genau wie bei einem guten alten Array.

```
new ArrayList<Song>()
```

### 2 Deklaration und Zuweisung von *Variablen* generischer Typen

Wie funktioniert Polymorphie genau bei generischen Typen? Wenn Sie eine ArrayList<Haustier>-Referenzvariable haben – können Sie ihr eine ArrayList<Hund> zuweisen? Was ist mit einer List<Haustier>-Referenz? Können Sie der eine ArrayList<Haustier> zuweisen? Sie werden es sehen ...

```
List<Song> songList =
    new ArrayList<Song>()
```

### 3 Deklaration (und Aufruf) von *Methoden*, denen generische Typen übergeben werden

Wenn eine Methode beispielsweise eine ArrayList von Haustier-Objekten entgegennimmt – was bedeutet das genau? Können Sie ihr auch eine ArrayList von Hund-Objekten übergeben? Wir sehen uns dazu ein paar knifflige Polymorphie-Probleme an, die sich sehr von der Art und Weise unterscheiden, in der man Methoden schreibt, die Arrays entgegennehmen.

(Eigentlich geht es hier um das Gleiche wie in Punkt 2, aber das zeigt Ihnen, für wie wichtig wir es halten.)

```
void foo(List<Song> liste)

x.foo(songList)
```

**F: Aber muss ich nicht auch lernen, wie ich meine EIGENEN generischen Klassen erzeuge? Was ist, wenn ich eine Klasse schreiben will, bei der diejenigen, die die Klasse instantiieren, selbst entscheiden können, welchen Typ von Dingen diese Klasse verwenden soll?**

A: Das werden Sie wahrscheinlich nicht sehr oft tun. Überlegen Sie mal – die API-Designer haben eine ganze Bibliothek von Collection-Klassen erstellt, mit denen die meisten Datenstrukturen, die Sie brauchen, abgedeckt sind; und die einzigen Klassen, die wirklich generisch sein müssen, sind eigentlich die Collection-Klassen. Also Klassen, die andere Elemente enthalten sollen und bei denen Sie möchten, dass andere Programmierer beim Deklarieren und Instantiieren der Collection-Klasse genau festlegen, von welchem Typ diese Elemente sein sollen.

Es ist tatsächlich denkbar, dass Sie generische Klassen *erzeugen* wollen, aber da das eine Ausnahme sein wird, behandeln wir es hier nicht. (Mit Hilfe dessen, was wir hier besprechen, werden Sie es auch selbst herausfinden.)

# Verwendung von generischen KLASSEN

Da ArrayList unser meistgenutzter generischer Typ ist, beginnen wir mit einem Blick in die ArrayList-Dokumentation. Die beiden wichtigsten Stellen, die man sich bei einer generischen Klasse ansehen muss, sind:

1) die Deklaration der *Klasse* und

2) die Deklaration der *Methoden*, mit denen Sie Elemente hinzufügen können

Betrachten Sie »E« als Platzhalter für »den Elementtyp, den diese Collection enthalten und zurückgeben soll«. (E steht für Elementtyp.)

## Die ArrayList-Dokumentation verstehen

### *(Oder: Was sich wirklich hinter »E« verbirgt)*

Das »E« ist ein Platzhalter für den EIGENTLICHEN Typ, den Sie bei der Deklaration und Erzeugung einer ArrayList verwenden.

ArrayList ist eine Unterklasse von AbstractList; der Typ, den Sie für die ArrayList spezifizieren, wird also automatisch auch für den Typ der AbstractList benutzt.

```
public class ArrayList<E> extends AbstractList<E> implements List<E> ... {

      public boolean add(E o)

      // mehr Code
}
```

Hier der wichtige Teil! Was immer »E« auch ist, es legt fest, welche Art von Dingen Sie zu der ArrayList hinzufügen dürfen.

Der Typ (der Wert von <E>) wird auch zum Typ des List-Interface.

Das »E« repräsentiert den Typ, der zur Erzeugung einer ArrayList-Instanz benutzt wird. Wenn Sie ein »E« in der ArrayList-Dokumentation sehen, können Sie es in Gedanken mit Suchen/Ersetzen gegen den <Typ> austauschen, mit dem Sie die ArrayList instantiieren wollen.

new ArrayList<Song> bedeutet also: In jeder Methoden- oder Variablendeklaration, in der »E« benutzt wird, wird »E« durch »Song« ersetzt.

# Verwendung von Typparametern bei ArrayList

**DIESER Code:**

```
ArrayList<String> dieseList = newArrayList<String>
```

**bedeutet, dass ArrayList**

```
public class ArrayList<E> extends AbstractList<E> ... {

      public boolean add(E o)
      // weiterer Code
}
```

**vom Compiler behandelt wird als:**

```
public class ArrayList<String> extends AbstractList<String> ... {

      public boolean add(String o)
      // weiterer Code
}
```

Mit anderen Worten, das »E« wird durch den *tatsächlichen* Typ (der auch als *Typparameter* bezeichnet wird) ersetzt, den Sie bei der Erzeugung der ArrayList benutzen. Und das ist der Grund dafür, dass die ArrayList-Methode add() Sie nichts anderes hinzufügen lässt als Objekte eines Referenztyps, der mit dem Typ von »E« kompatibel ist. Wenn Sie also eine ArrayList**<String>** erzeugen, wird aus der Methode add() plötzlich **add(String o)**. Und wenn Sie für die ArrayList den Typ **Hund** festlegen, wird aus add() plötzlich **add(Hund o)**.

**F: Ist »E« das Einzige, was man dort einsetzen kann? Bei der sort-Methode stand in den Javadocs doch »T « ...**

A: Sie können alles verwenden, was ein zulässiger Java-Identifizierer ist, d.h., alles, was Sie für einen Methoden- oder Variablennamen verwenden könnten, würde als Typparameter funktionieren. Konventionsgemäß verwendet man jedoch nur einen einzelnen Buchstaben (also sollten Sie das auch tun). Nach einer weiteren Konvention sollte man »T« verwenden – außer bei Collection-Klassen, dort verwendet man »E«, das für den »**E**lementtyp in der Collection« steht.

# Verwendung von generischen METHODEN

Bei einer generischen *Klasse* enthält die *Klassendeklaration* einen Typparameter. Bei einer generischen *Methode* wird bei der Deklaration ein Typparameter in der Signatur angegeben.

Sie können Typparameter in einer Methode auf unterschiedliche Art verwenden:

### ① Verwendung eines in der Klassendeklaration definierten Typparameters

```
public class ArrayList<E> extends AbstractList<E> ... {
    public boolean add(E o)
```

Sie können das »E« hier NUR benutzen, weil es schon als Teil der Klasse definiert worden ist.

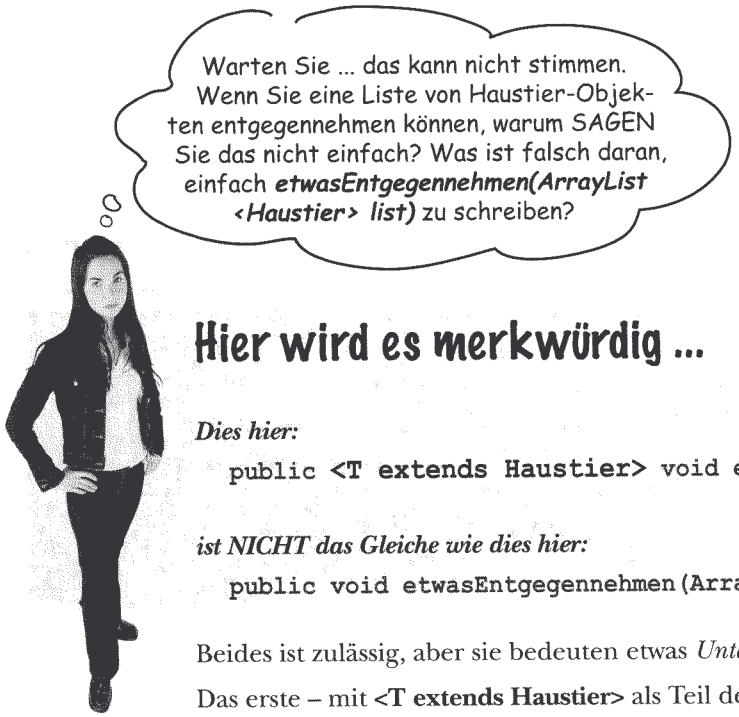
Wenn Sie einen Typparameter für die Klasse deklarieren, können Sie diesen Typ einfach überall dort einsetzen, wo Sie als Typ eine *wirkliche* Klasse oder ein *wirkliches* Interface benutzt hätten. Der im Methodenargument deklarierte Typ wird dann letztendlich durch den Typ ersetzt, den Sie beim Instantiieren der Klasse verwenden.

### ② Verwendung eines NICHT in der Klassendeklaration definierten Typparameters

```
public <T extends Haustier> void etwasEntgegennehmen(ArrayList<T> liste)
```

Hier können wir <T> benutzen, weil wir »T« vorne in der Deklaration der Methode schon deklariert haben.

Wenn die Klasse selbst keinen Typparameter verwendet, können Sie trotzdem noch einen für eine Methode angeben, indem Sie ihn an einer sehr ungewöhnlichen (aber gültigen) Stelle deklarieren: *vor dem Rückgabetyp*. In dieser Methode kann T beispielsweise »jeder Typ von Haustier« sein.




# Hier wird es merkwürdig ...

*Dies hier:*

```
public <T extends Haustier> void etwasEntgegennehmen(ArrayList<T> liste)
```

*ist NICHT das Gleiche wie dies hier:*

```
public void etwasEntgegennehmen(ArrayList<Haustier> liste)
```

Beides ist zulässig, aber sie bedeuten etwas *Unterschiedliches!*

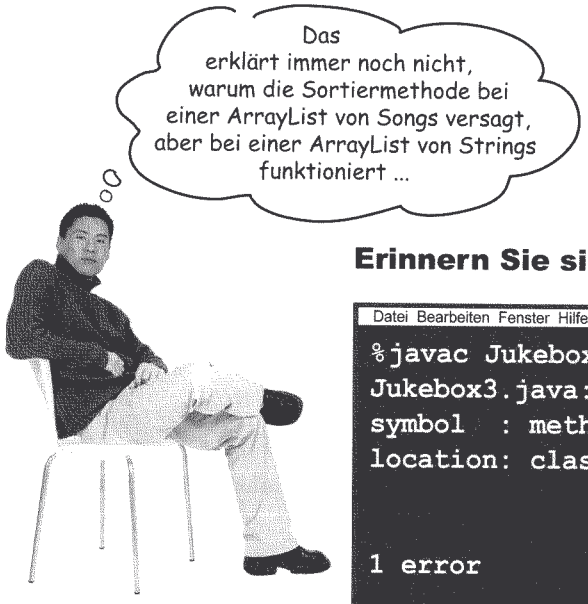
Das erste – mit **<T extends Haustier>** als Teil der Methodendeklaration – bedeutet, dass jede ArrayList zulässig ist, die mit dem Typ Haustier oder einem Untertyp von Haustier (wie Hund oder Katze) deklariert wurde. Daher könnten Sie die obere Methode mit ArrayList<Hund>, ArrayList<Katze> oder ArrayList<Haustier> aufrufen.

Die untere Methode – mit dem Argument (ArrayList<Haustier> liste) – bedeutet dagegen, dass *nur* eine ArrayList<Haustier> zulässig ist. Anders ausgedrückt: Während die erste Version eine ArrayList von jedem Typ annimmt, der ein Haustier-Typ ist (Haustier, Hund, Katze usw.), nimmt die zweite Version *ausschließlich* eine ArrayList vom Typ Haustier an. Keine ArrayList<Hund>, keine ArrayList<Katze>, sondern nur ArrayList<Haustier>.

Ja, das scheint tatsächlich gegen die Grundsätze der Polymorphie zu verstoßen, aber wenn wir am Ende dieses Kapitels noch einmal darauf zurückkommen, werden Sie es verstehen. Erinnern Sie sich, im Moment sehen wir uns das ja nur an, weil wir noch immer herausfinden wollen, wie man diese SongList sortieren kann. Und deshalb hatten wir in der API unter der Methode sort() nachgeschaut, die diese merkwürdige Deklaration mit dem generischen Typ hatte.

***Vorläufig müssen Sie nur wissen, dass die Syntax der oberen Version zulässig ist und dass sie bedeutet: Sie können ein ArrayList-Objekt übergeben, das als Haustier oder irgendein Haustier-Untertyp instantiiert wird.***

Und jetzt zurück zu unserer Methode sort() ...




## Erinnern Sie sich, wo wir stehen geblieben waren ...

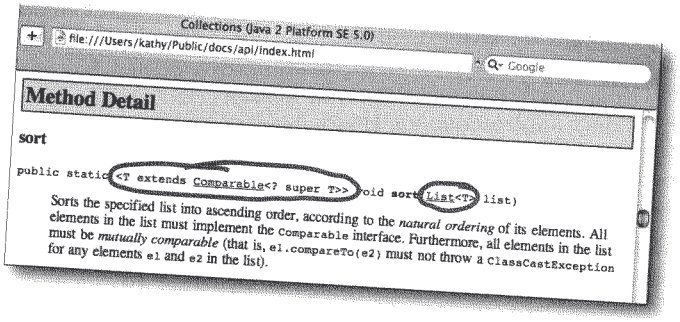
```
Datei Bearbeiten Fenster Hilfe Bums
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol  : method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
                Collections.sort(songList);
                           ^
1 error
```

```
import java.util.*;
import java.io.*;

public class Jukebox3 {
  ArrayList<Song> songList = new ArrayList<Song>();
  public static void main(String[] args) {
    new Jukebox3().los();
  }
  public void los() {
    getSongs();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);
  }
  void getSongs() {
    try {
      File datei = new File("SongListPlus.txt");
      BufferedReader reader = new BufferedReader(new FileReader(datei));
      String zeile = null;
      while ((zeile= reader.readLine()) != null) {
        addSong(zeile);
      }
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
  void addSong(String zuParsendeZeile) {
    String[] tokens = zuParsendeZeile.split("/");
    Song nächsterSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
    songList.add(nächsterSong);
  }
}
```

An dieser Stelle funktioniert es nicht mehr! Beim Übergeben einer ArrayList<Strings> hat es wunderbar funktioniert, aber beim Versuch, eine ArrayList<Song> zu sortieren, sind wir gescheitert.

# Noch einmal zurück zur Methode sort()

Da sind wir also wieder und versuchen die Docs für die Methode sort() zu lesen, um herauszufinden, warum das Sortieren bei einer Liste von Strings okay war und bei einer Liste von Song-Objekten nicht. Und die Antwort ist anscheinend ...




**Die Methode sort() kann nur Listen von Comparable-Objekten entgegennehmen.**

**Song ist KEIN Untertyp von Comparable, daher können Sie die Liste von Songs nicht sortieren.**

***Zumindest NOCH nicht ...***

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Dies bedeutet: »Was T auch sein mag – es muss vom Typ Comparable sein.«

(Ignorieren Sie diesen Teil im Moment. Falls Sie das nicht können: Es bedeutet einfach, dass der Typparameter für Comparable dem Typ T oder einem Supertyp von T entsprechen muss.)

Sie können nur eine List (oder einen Untertyp von List, wie ArrayList) mit einem parametrisierten Typ übergeben, der »Comparable« erweitert.

Ähm ... ich habe gerade in den Docs bei String nachgesehen, und String ERWEITERT Comparable nicht – es IMPLEMENTIERT es. **Comparable ist ein Interface.** Es ist also Unsinn, <T ***extends*** Comparable> zu schreiben.

```
public final class String extends Object implements Serializable,
                                    Comparable<String>, CharSequence
```

# »extends« bedeutet bei Generics »extends oder implements«

Die Java-Entwickler ermöglichen es Ihnen, eine Einschränkung für einen parametrisierten Typ anzugeben, damit Sie ihn beispielsweise auf Unterklassen von Haustier begrenzen können. Aber Sie müssen einen Typ auch in einer Weise einschränken können, dass nur Klassen erlaubt sind, die ein bestimmtes Interface implementieren. Hier haben wir also eine Situation, in der wir *eine* Syntax für beide Situationen brauchen – Vererbung und Implementierung. Mit anderen Worten eine Syntax, die sowohl für *extends* als auch für *implements* funktioniert.

Und der Gewinner war das Wort ... *extends*. Aber eigentlich bedeutet es hier »ist-ein« und funktioniert unabhängig davon, ob der Typ auf der rechten Seite ein Interface oder eine Klasse ist.

**Bei Generics bedeutet das Schlüsselwort »extends« eigentlich »ist-ein«, und es funktioniert sowohl bei Klassen als auch bei Interfaces.**

Comparable ist ein Interface, also liest man das EIGENTLICH so: »T muss ein Typ sein, der das Interface Comparable implementiert.«

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Es spielt keine Rolle, ob das Ding rechts eine Klasse oder ein Interface ist ... es heißt in jedem Fall »extends«.

**F: Warum hat man nicht einfach ein neues Schlüsselwort »is« eingeführt?**

A: Das Hinzufügen eines neuen Schlüsselworts zu einer Sprache ist WIRKLICH eine große Sache, weil damit riskiert wird, Java-Code unbrauchbar zu machen, den Sie in einer früheren Version geschrieben haben. Überlegen Sie – vielleicht verwenden Sie irgendwo eine Variable »is« (was wir in diesem Buch tatsächlich tun, nämlich für InputStreams). Und da Sie Schlüsselwörter nicht als Identifizierer in Ihrem Code verwenden dürfen, bedeutet das: Der gesamte Code, der das Schlüsselwort verwendet hat, *bevor* es ein reserviertes Wort wurde, funktioniert nicht mehr. Wenn also die Entwickler bei Sun die Chance haben, ein vorhandenes Schlüsselwort zu benutzen, werden sie sich normalerweise immer *dafür* entscheiden, so wie hier für »extends«. Manchmal haben sie allerdings keine Wahl ...

Einige (sehr wenige) neue Schlüsselwörter *sind* in die Sprache aufgenommen worden, z.B. ***assert*** mit Java 1.4 und ***enum*** mit Java 5.0 (enum sehen wir uns im Anhang an). Und das macht den Leuten tatsächlich den Code kaputt – allerdings hat man manchmal die Möglichkeit, Code mit einer *neueren* Java-Version so zu kompilieren und auszuführen, dass diese sich wie eine ältere Version verhält. Dazu übergeben Sie dem Compiler oder der JVM auf der Kommandozeile einen speziellen Schalter, der signalisiert: »Ja, ja, ich WEISS, dass dies Java 1.4 ist, aber bitte tu so, als wäre es 1.3, denn ich verwende in meinem Code eine Variable namens *assert*, die ich schon damals geschrieben habe, als ihr Jungs noch behauptet habt, es wäre okay!«

(Um herauszufinden, ob Ihnen ein solcher Schalter zur Verfügung steht, geben Sie auf der Kommandozeile *javac* (für den Compiler) oder *java* (für die JVM) ein, ohne irgendetwas dahinter. Sie sollten dann eine Liste der verfügbaren Optionen erhalten. Mehr über diese Schalter erfahren Sie im Kapitel über Deployment.)

# Endlich wissen wir, was verkehrt ist ...
## Die Klasse Song muss Comparable implementieren

Wir können die ArrayList<Song> nur dann der Methode sort() übergeben, wenn die Klasse Song Comparable implementiert, denn die Methode sort() ist in dieser Weise deklariert. Wie ein rascher Blick in die API-Docs zeigt, ist das Interface Comparable wirklich einfach; es muss nur eine einzige Methode implementiert werden:

**java.lang.Comparable**

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

Und in der Methodendokumentation für compareTo() heißt es:

**Returns:**
**a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.**

Anscheinend wird die Methode compareTo() auf einem Song-Objekt aufgerufen, und dabei wird diesem Song eine Referenz auf einen anderen Song übergeben. Der Song, der die Methode ausführt, muss herausfinden, ob der ihm übergebene Song weiter oben, weiter unten oder an der gleichen Stelle in der Liste einsortiert wird wie er selbst.

Ihre große Aufgabe ist es jetzt, zu entscheiden, was einen Song größer macht als einen anderen, und dann die Methode compareTo() entsprechend zu implementieren. Eine negative Zahl (irgendeine) bedeutet, dass der übergebene Song größer ist als der Song, auf dem die Methode ausgeführt wird. Wird eine positive Zahl zurückgegeben, heißt das, dass der Song, auf dem die Methode ausgeführt wird, größer ist als der übergebene Song. Ein Rückgabewert 0 bedeutet, dass die Songs gleich sind (zumindest für Sortierzwecke; das heißt nicht unbedingt, dass beide das gleiche Objekt sind). Es könnten beispielsweise zwei Songs mit dem gleichen Titel sein.

(Was uns wiederum vor einen ganzen Haufen neuer Probleme stellt, die wir uns später ansehen werden ...)

**Die große Frage ist: Was macht *einen* Song kleiner als, gleich oder größer als einen *anderen* Song?**

**Sie können das Interface Comparable erst dann implementieren, wenn Sie das beantwortet haben.**

### Spitzen Sie Ihren Bleistift

Schreiben Sie Ihre Ideen und den Pseudocode (oder noch besser RICHTIGEN Code) für eine Implementierung der compareTo()-Methode auf, die die Song-Objekte nach dem Titel sortieren würde.

Hinweis: Wenn Sie auf der richtigen Spur sind, sollten es weniger als drei Zeilen Code sein!

# Die neue, verbesserte, vergleichbare Song-Klasse

Wir haben uns dafür entschieden, nach Titel zu sortieren, daher implementieren wir die Methode compareTo() so, dass der Titel des Songs, der unserer Methode übergeben wird, mit dem Titel des Songs verglichen wird, auf dem die Methode aufgerufen wurde. Anders ausgedrückt: Der Song, der die Methode ausführt, muss entscheiden, was der Vergleich seines Titels mit dem Titel des Methodenparameters ergibt.

Hmmm ... wir wissen, dass die Klasse String die alphabetische Reihenfolge kennen muss, denn die Methode sort() hat mit einer Liste von Strings funktioniert. Wir wissen auch, dass String eine Methode compareTo() hat – warum rufen wir diese also nicht einfach auf? So kann sich ganz leicht ein Titel-String mit einem anderen vergleichen, und wir müssen den Algorithmus für den Vergleich und das alphabetische Sortieren nicht selbst schreiben!

Normalerweise stimmen diese beiden überein ... wir geben hier den Typ an, mit dem die implementierende Klasse verglichen werden kann.
Dies bedeutet, dass Song-Objekte für Sortierzwecke mit anderen Song-Objekten verglichen werden können.

```
class Song implements Comparable<Song> {
  String titel;
  String künstler;
  String bewertung;
  String bpm;

  public int compareTo(Song s) {
    return titel.compareTo(s.getTitel());
  }

  Song(String t, String k, String bw, String b) {
    titel = t;
    künstler = k;
    bewertung = bw;
    bpm = b;
  }

  public String getTitel() {
    return titel;
  }

  public String getKünstler() {
    return künstler;
  }

  public String getBewertung() {
    return bewertung;
  }

  public String getBpm() {
    return bpm;
  }

  public String toString() {
    return titel;
  }
}
```

Die Methode sort() schickt einen Song zum Vergleichen an compareTo(), um herauszufinden, was ein Vergleich dieses Songs mit dem Song ergibt, auf dem die Methode aufgerufen wurde.

Einfach! Wir geben die Arbeit lediglich an die Titel-String-Objekte weiter, denn wir wissen, dass Strings eine compareTo()-Methode haben.

Dieses Mal hat es funktoniert. Die Liste wird ausgegeben, dann wird sort() aufgerufen, wodurch die Songs alphabetisch nach dem Titel sortiert werden.

```
Datei Bearbeiten Fenster Hilfe Ambient
%java Jukebox3

[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]
```

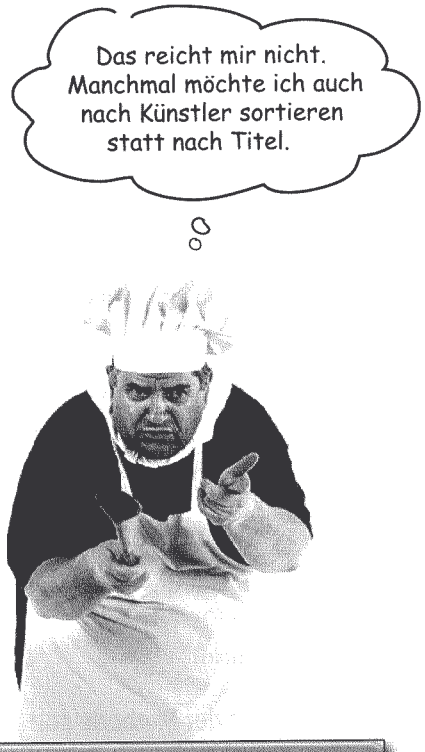
# Wir können die Liste sortieren, aber ...

Wir haben ein neues Problem: Jupp möchte zwei unterschiedliche Ansichten der Songliste – eine nach dem Songtitel und eine nach dem Künstler sortiert!

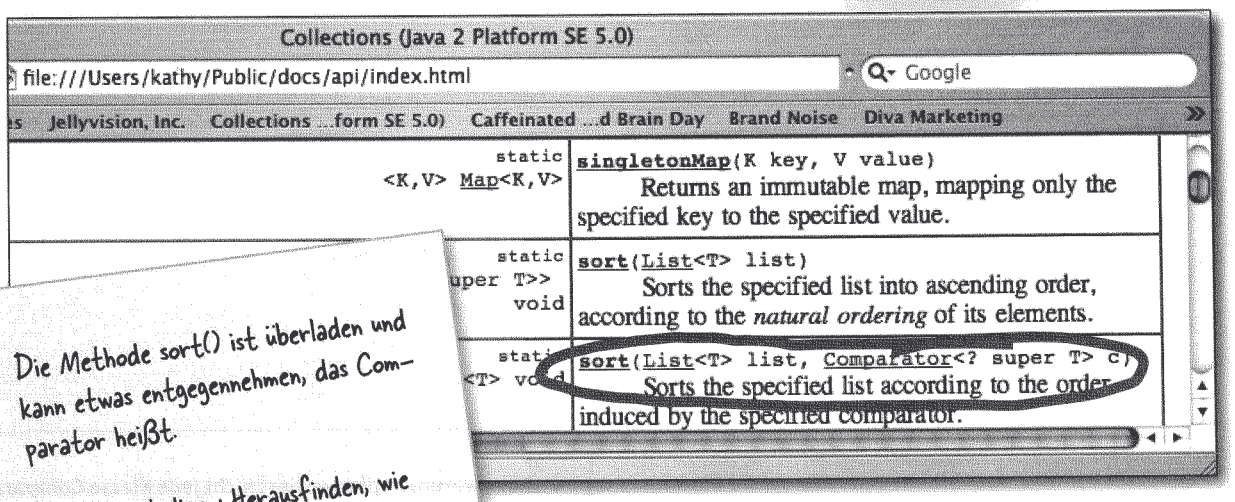
Aber wenn Sie ein Collection-Element vergleichbar machen (indem Sie es Comparable implementieren lassen), können Sie die Methode compareTo() nur ein einziges Mal implementieren. Was können Sie also tun?

Der hässliche Weg wäre die Einführung einer Schaltervariablen in der Klasse Song und eines entsprechenden *if*-Tests in compareTo(). Abhängig davon, ob der Schalter auf den Vergleich nach Titel oder nach Künstler eingestellt ist, würde dann ein unterschiedliches Ergebnis zurückgegeben.

Aber das ist eine scheußliche und anfällige Lösung – da gibt es etwas viel Besseres. Etwas, das nur für den Fall, dass Sie ein und dasselbe Ding auf mehr als eine Weise sortieren möchten, in die API eingebaut wurde.




**Sehen Sie sich noch einmal die Klasse Collections in der API an. Es gibt eine zweite sort()-Methode – und der wird ein Comparator übergeben.**




Die Methode sort() ist überladen und kann etwas entgegennehmen, das Comparator heißt.

Noch zu erledigen: Herausfinden, wie man einen Comparator bekommt/erzeugt, der die Songs nach dem Künstler statt nach dem Titel vergleicht und sortiert ...

# Einen selbst definierten Comparator verwenden

Ein Element in einer Liste kann *sich selbst* auf nur *eine* Art mit einem anderen Exemplar des eigenen Typs vergleichen, nämlich mit Hilfe seiner compareTo()-Methode. Ein Comparator befindet sich dagegen außerhalb des Elements, das verglichen wird – er ist eine separate Klasse. Sie können also so viele davon machen, wie Sie wollen! Sie wollen Songs nach dem Künstler vergleichen? Machen Sie einen KünstlerComparator. Nach Schlägen pro Minute (BPM)? Machen Sie einen BPMComparator.

Dann müssen Sie nur noch die überladene sort()-Methode aufrufen und ihr die Liste und den Comparator übergeben, mit dessen Hilfe die sort()-Methode die Dinge in die richtige Reihenfolge bringen kann.

Die sort()-Methode, die einen Comparator erwartet, verwendet beim Sortieren der Elemente den Comparator statt der elementeigenen compareTo()-Methode. Anders ausgedrückt: Wenn Ihre sort()-Methode einen Comparator erhält, wird sie die compareTo()-Methode der Elemente in der Liste noch nicht einmal *aufrufen*. Stattdessen ruft die sort()-Methode die Methode **compare()** auf dem Comparator auf.

Die Regeln sind also:

- **Ein Aufruf der Methode sort(List o) (mit einem Argument) bedeutet, dass die compareTo()-Methode des Listenelements die Reihenfolge bestimmt. Die Elemente in der Liste MÜSSEN daher das Interface Comparable implementieren.**
- **Ein Aufruf von sort(List o, Comparator c) bedeutet, dass die compareTo()-Methode des Listenelements NICHT aufgerufen wird und dass stattdessen die compare()-Methode des Comparators benutzt wird. Das heißt, dass die Elemente in der Liste NICHT das Interface Comparable implementieren müssen.**

**java.util.Comparator**

```
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

Wenn Sie der sort()-Methode einen Comparator übergeben, wird die Sortierreihenfolge durch den Comparator bestimmt statt durch die elementeigene compareTo()-Methode.

---

**F: Bedeutet das also, dass Sie bei einer Klasse, die Comparable nicht implementiert und deren Quellcode Sie nicht haben, trotzdem noch alles in die richtige Reihenfolge bringen könnten, wenn Sie einen Comparator erzeugten?**

A: Das ist richtig. Die andere Möglichkeit (falls zulässig) wäre, eine Unterklasse des Elements zu schreiben und diese Comparable implementieren zu lassen.

**F: Aber warum implementiert nicht *jede* Klasse Comparable?**

A: Glauben Sie wirklich, dass man *alles* sortieren kann? Bei Elementtypen, die nicht von sich aus irgendeine Art von natürlicher Ordnung anbieten, würden Sie andere Programmierer nur verwirren, wenn Sie Comparable implementierten. Und wenn Sie Comparable nicht implementieren, gehen Sie auch kein großes Risiko ein, denn ein Programmierer kann ja alles auf jede beliebige Art miteinander vergleichen, indem er seinen eigenen Comparator schreibt.

# Jukebox-Update mit einem Comparator

Drei Dinge haben wir in diesem Code gemacht:

1) Eine innere Klasse erzeugt, die Comparator implementiert (und damit die Methode ***compare()***, die die bisher von ***compareTo()*** erledigte Arbeit übernimmt).

2) Eine Instanz der inneren Comparator-Klasse erzeugt.

3) Die überladene sort()-Methode aufgerufen und ihr sowohl die Songliste als auch die Instanz der inneren Comparator-Klasse übergeben.

Beachten Sie: Wir haben auch die toString()-Methode der Klasse Song aktualisiert; sie gibt jetzt sowohl den Songtitel als auch den Künstler aus. (Es wird ***Titel: Künstler*** ausgegeben – egal, wonach die Liste sortiert ist.)

```
import java.util.*;
import java.io.*;

public class Jukebox5 {
  ArrayList<Song> songList = new ArrayList<Song>();
  public static void main(String[] args) {
    new Jukebox5().los();
  }

  class KünstlerVergleich implements Comparator<Song> {
    public int compare(Song eins, Song zwei) {
      return eins.getKünstler().compareTo(zwei.getKünstler());
    }
  }

  public void los() {
    getSongs();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);

    KünstlerVergleich künstlerVergleich = new KünstlerVergleich();
    Collections.sort(songList, künstlerVergleich);

    System.out.println(songList);
  }

  void getSongs() {
    // E/A-Code
  }

  void addSong(String zuParsendeZeile) {
    // Zeile parsen und zur Songliste hinzufügen
  }
}
```

Wir erzeugen eine neue innere Klasse, die Comparator implementiert (beachten Sie, dass der Typparameter dem Typ entspricht, den wir vergleichen werden – in diesem Fall Song-Objekte).

Dies wird ein String (der Künstler).

Wir überlassen den String-Variablen (für die Künstler) das eigentliche Vergleichen, da Strings schon wissen, wie sie sich alphabetisch anordnen.

Wir erzeugen eine Instanz der inneren Comparator-Klasse.

Wir rufen sort() auf und übergeben ihr die Liste und eine Referenz auf das neue, selbst gemachte Comparator-Objekt.

Anmerkung: Wir haben die Sortierung nach Titel als Standard vorgegeben, indem wir die compareTo()-Methode in Song weiterhin nach Titel sortieren lassen. Alternativ könnten wir hier sowohl die Titel- als auch die Künstlersortierung als innere Comparator-Klassen implementieren und die Implementierung von Comparable bei Song streichen. Das hieße, dass wir ausschließlich mit der Zwei-Argument-Version von Collections.sort() arbeiten würden.

```
import _____;
public class BergSortierer {
  LinkedList_____ berge = new LinkedList_____();
  class Namensvergleich _____ {
    public int compare(Berg eins, Berg zwei) {
      return _____;
    }
  }
  class Höhenvergleich _____ {
    public int compare(Berg eins, Berg zwei) {
      return (_____);
    }
  }
  public static void main(String [] args) {
    new BergSortierer().los();
  }
  public void los() {
    berge.add(new Berg("Matterhorn", 4478));
    berge.add(new Berg("Dom", 4545));
    berge.add(new Berg("Nordend", 4609));
    berge.add(new Berg("Nadelhorn", 4327));

    System.out.println("Wie eingegeben:\n" + berge);
    Namensvergleich nc = new Namensvergleich();
    _____;
    System.out.println("Nach Namen:\n" + berge);
    Höhenvergleich hc = new Höhenvergleich();
    _____;
    System.out.println("Nach Höhe:\n" + berge);
  }
}
class Berg {
  _____;
  _____;
  _____ {
    _____;
    _____;
  }
  _____ {
    _____ ;
  }
}
```

## Reverse Engineering

Nehmen Sie an, dieser Code stünde in einer einzigen Datei. Ihre Aufgabe ist es, die Lücken so auszufüllen, dass das Programm die gezeigte Ausgabe erzeugt.

Anmerkung: Die Lösung finden Sie am Ende dieses Kapitels.

**Ausgabe:**

```
Datei Bearbeiten Fenster Hilfe BergeVersetzen
% java BergSortierer
Wie eingegeben:
[Matterhorn 4478, Dom 4545, Nordend 4609, Nadelhorn 4327]
Nach Namen:
[Dom 4545, Matterhorn 4478, Nadelhorn 4327, Nordend 4609]
Nach Höhe:
[Nordend 4609, Dom 4545, Matterhorn 4478, Nadelhorn 4327]
```

## Lückentext

Tragen Sie hinter den Fragen als Antwort jeweils eins der Wörter aus der Liste der »möglichen Antworten« ein. Die Lösung finden Sie am Ende dieses Kapitels.

**Mögliche Antworten:**

**Comparator**

**Comparable**

**compareTo()**

**compare()**

**ja**

**nein**

Gegeben ist folgende kompilierbare Anweisung:

```
Collections.sort(meineArrayList);
```

1. Was muss die Klasse der in `meineArrayList` gespeicherten Objekte implementieren? _____
2. Welche Methode muss die Klasse der in `meineArrayList` gespeicherten Objekte implementieren? _____
3. Kann die Klasse der in `meineArrayList` gespeicherten Objekte sowohl Comparator als auch Comparable implementieren? _____

Gegeben ist folgende kompilierbare Anweisung:

```
Collections.sort(meineArrayList, meinVergleich);
```

4. Kann die Klasse der in `meineArrayList` gespeicherten Objekte `Comparable` implementieren? _____
5. Kann die Klasse der in `meineArrayList` gespeicherten Objekte `Comparator` implementieren? _____
6. Muss die Klasse der in `meineArrayList` gespeicherten Objekte `Comparable` implementieren? _____
7. Muss die Klasse der in `meineArrayList` gespeicherten Objekte `Comparator` implementieren? _____
8. Was muss die Klasse des Objekts `meinVergleich` implementieren? _____
9. Welche Methode muss die Klasse des Objekts `meinVergleich` implementieren? _____

# O-oh! Die Sortierung funktioniert, aber jetzt haben wir welche doppelt ...

Die Sortierung klappt jetzt wunderbar, da wir wissen, wie man sowohl nach *Titel* sortiert (mit der compareTo()-Methode des Song-Objekts) als auch nach *Künstler* (mit der compare()-Methode des Comparators). Aber es gibt ein neues Problem, das wir mit einem Testauszug aus der Jukebox-Textdatei nicht bemerkt hatten – ***die sortierte Liste enthält Elemente mehrfach.***

Anscheinend schreibt die Restaurant-Jukebox einfach immer weiter in die Datei, egal ob ein Stück schon einmal gespielt (und somit geschrieben) wurde oder nicht. Die Jukebox-Textdatei SongListPlus.txt ist eine vollständige Aufzeichnung aller gespielten Songs, daher kann sie den gleichen Song mehrfach enthalten.

```
Datei Bearbeiten Fenster Hilfe ZuVieleTöne
% java JukeBox5

[Pink Moon: Nick Drake, Somersault: Zero 7, Shiva Moon: Prem
Joshua, Circles: BT, Deep Channel: Afro Celts, Passenger:
Headmix, Listen: Tahiti 80, Listen: Tahiti 80, Listen: Tahiti
80, Circles: BT]

[Circles: BT, Circles: BT, Deep Channel: Afro Celts, Listen:
Tahiti 80, Listen: Tahiti 80, Listen: Tahiti 80, Passenger:
Headmix, Pink Moon: Nick Drake, Shiva Moon: Prem Joshua,
Somersault: Zero 7]

[Deep Channel: Afro Celts, Circles: BT, Circles: BT, Passenger:
Headmix, Pink Moon: Nick Drake, Shiva Moon: Prem Joshua, Listen:
Tahiti 80, Listen: Tahiti 80, Listen: Tahiti 80, Somersault:
Zero 7]
```

Vor der Sortierung.

Nach der Sortierung mit Hilfe der Song-eigenen compareTo()-Methode (Sortierung nach Titel).

Nach der Sortierung mit Hilfe des KünstlerVergleich-Comparators (Sortierung nach Künstlername).

**SongListPlus.txt**

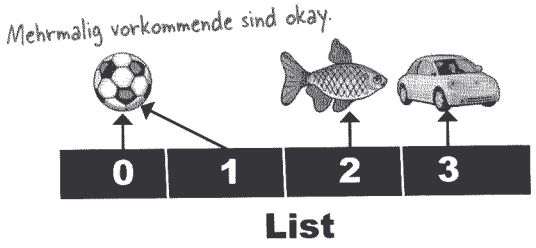
```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Circles/BT/5/110
```

In der Textdatei SongListPlus stehen jetzt manche Songs mehrmals, da der Jukebox-Automat alle Songs aufzeichnet, und zwar in der Reihenfolge, in der sie gespielt werden. Irgendjemand hat beschlossen, »Listen« dreimal hintereinander zu spielen, und darauf folgte »Circles«, ein Song, der vorher schon einmal gespielt worden war.

Wir können die Art und Weise, wie die Textdatei geschrieben wird, nicht ändern, denn manchmal werden wir all diese Informationen brauchen. Wir müssen den Java-Code ändern.

# Wir brauchen ein Set statt einer List

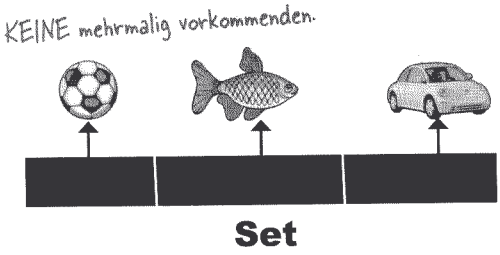
In der Collection-API finden wir drei grundlegende Interfaces, **List, Set** und **Map.** ArrayList ist eine **List,** aber offenbar ist ***Set*** genau das, was wir brauchen.




➤ **LIST – wenn es auf die *Reihenfolge* ankommt**

Collections, die die ***Indexposition*** kennen.

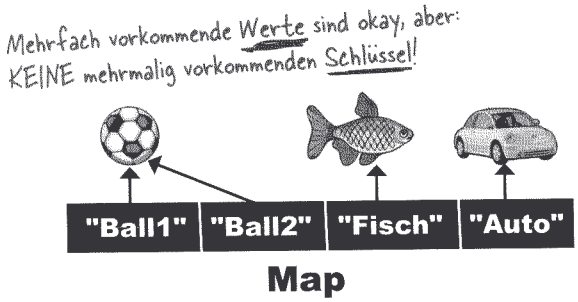
Lists wissen, wo etwas in der List steht. Das gleiche Objekt kann durch mehrere Elemente referenziert werden.




➤ **SET – wenn es auf die *Einzigartigkeit* ankommt**

Collections, die ***nichts mehrmals enthalten dürfen.***

Sets wissen, ob etwas schon in der Sammlung vorkommt. Es kann nie mehr als ein Element das gleiche Objekt referenzieren (und es können auch nicht zwei Objekte referenziert werden, die als gleich gelten; was Gleichheit bei Objekten bedeutet, sehen wir gleich noch).




➤ **MAP – wenn etwas anhand eines *Schlüssels gefunden* werden soll**

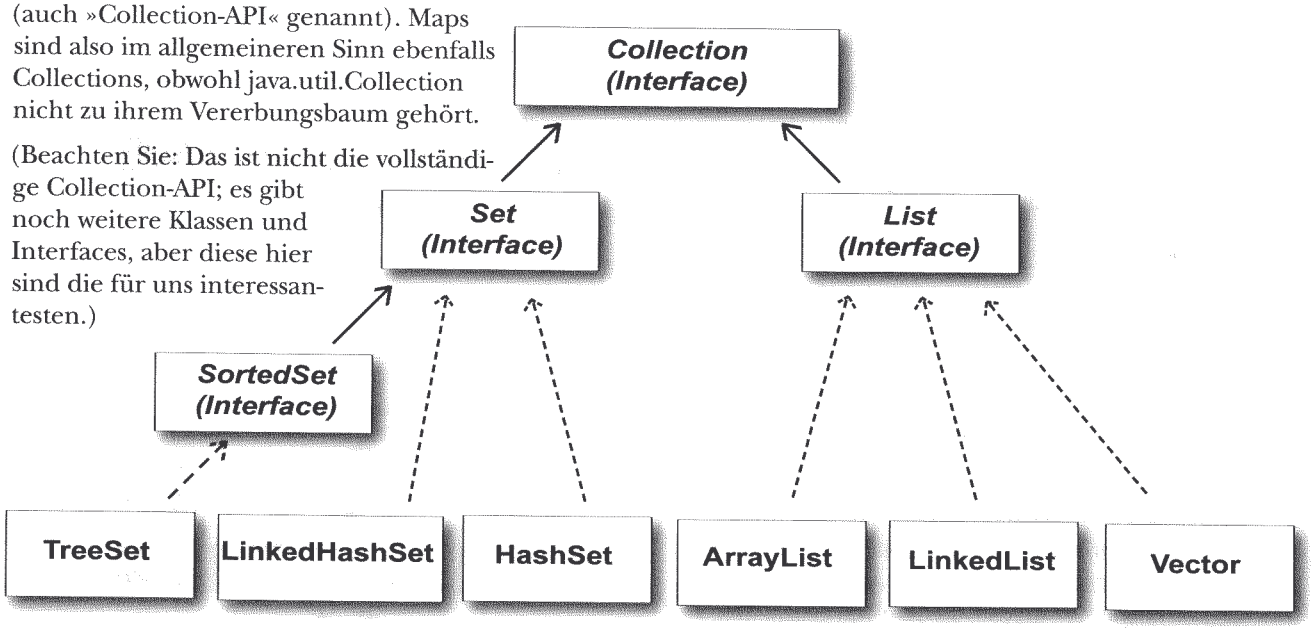
Collections, die ***Schlüssel/Wert-Paare*** verwenden.

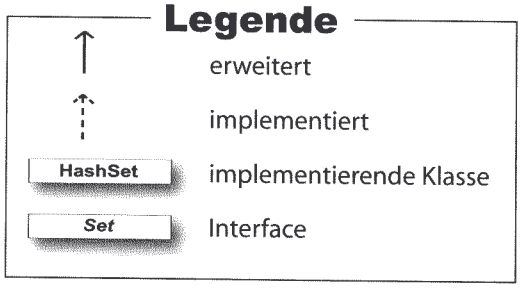
Maps wissen, welcher Wert mit einem bestimmten Schlüssel verknüpft ist. Es können zwei verschiedene Schlüssel vorkommen, die den gleichen Wert referenzieren, aber die Schlüssel selbst können nicht mehrmals vorkommen. Normalerweise werden als Schlüssel String-Namen benutzt (so dass Sie beispielsweise Name/Wert-Eigenschaftslisten erstellen können), aber ein Schlüssel darf jedes beliebige Objekt sein.

# Die Collection-API (ein Auszug)

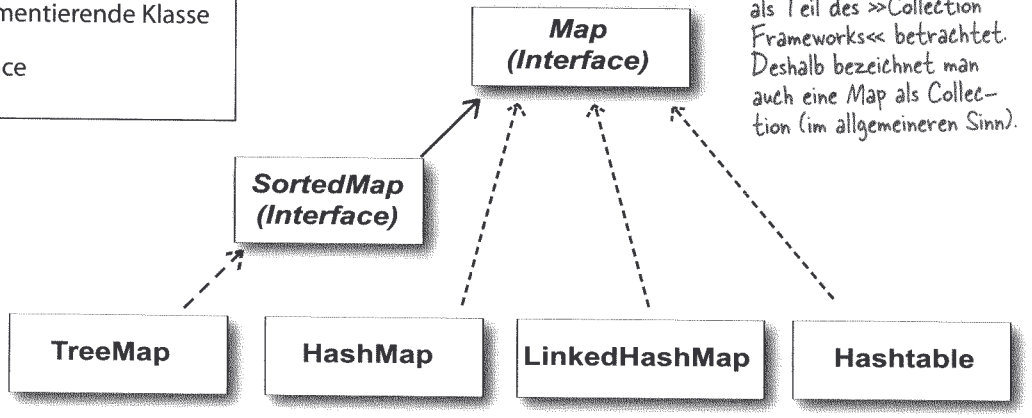
Das Interface Map ist, wie Sie sehen, eigentlich keine Erweiterung des Interface Collection. Dennoch betrachtet man es als Teil des »Collection Frameworks« (auch »Collection-API« genannt). Maps sind also im allgemeineren Sinn ebenfalls Collections, obwohl java.util.Collection nicht zu ihrem Vererbungsbaum gehört.

(Beachten Sie: Das ist nicht die vollständige Collection-API; es gibt noch weitere Klassen und Interfaces, aber diese hier sind die für uns interessantesten.)




Maps sind keine Erweiterung von java.util.Collection, sie werden aber trotzdem als Teil des »Collection Frameworks« betrachtet. Deshalb bezeichnet man auch eine Map als Collection (im allgemeineren Sinn).



# Umstieg von der ArrayList auf ein HashSet

Wir haben Code zur Jukebox hinzugefügt, der die Songs in ein HashSet setzt. (Anmerkung: Einen Teil des Jukebox-Codes haben wir weggelassen; Sie können ihn aus früheren Versionen kopieren. Und damit die Ausgabe besser lesbar ist, haben wir wieder die frühere Fassung der toString()-Methode von Song genommen, die nur den Titel statt Titel *und* Künstler ausgibt.)

```
import java.util.*;
import java.io.*;

public class Jukebox6 {
  ArrayList<Song> songList = new ArrayList<Song>();
  // main-Methode usw.

  public void los() {
    getSongs();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);
    HashSet<Song> songSet = new HashSet<Song>();
    songSet.addAll(songList);
    System.out.println(songSet);
  }
  // die Methoden getSongs() und addSongs()
}
```

Die Methode getSongs() haben wir nicht verändert, sie setzt die Songs nach wie vor in eine ArrayList.

Hier erzeugen wir ein neues HashSet, das so parametrisiert ist, dass es nur Songs enthält.

HashSet besitzt eine einfache Methode addAll(), die eine andere Collection entgegennimmt und damit das HashSet füllt. Das ist genauso, als würden wir alle Songs einzeln hinzufügen (aber viel einfacher).

```
Datei Bearbeiten Fenster Hilfe MitMusikGehtAllesBesser
% java Jukebox6

[Pink Moon, Somersault, Shiva Moon, Circles, Deep Channel,
Passenger, Listen, Listen, Listen, Circles]

[Circles, Circles, Deep Channel, Listen, Listen, Listen,
Passenger, Pink Moon, Shiva Moon, Somersault]

[Pink Moon, Listen, Shiva Moon, Circles, Listen, Deep Channel,
Passenger, Circles, Listen, Somersault]
```

Vor der Sortierung der ArrayList.

Nach der Sortierung der ArrayList (nach Titel).

Nachdem die Liste in ein HashSet gesetzt wurde, geben wir das HashSet aus.

Das Set hat nichts genützt! Wir haben immer noch all die doppelten!

(Und die Sortierreihenfolge ist auch verloren gegangen, als wir die List in ein HashSet gesteckt haben, aber darum kümmern wir uns später ...)

# Wann sind zwei Objekte gleich?

Erst einmal müssen wir fragen: Wann gelten zwei Song-Referenzen als doppelt vorkommend? Wenn Sie als ***gleich (equal)*** angesehen werden. Sind es einfach zwei Referenzen auf genau das gleiche Objekt? Oder auf zwei getrennte Objekte, die beide den gleichen *Titel* haben?

Damit kommen wir zu einem entscheidenden Problem: *Referenz*gleichheit vs. *Objekt*gleichheit.

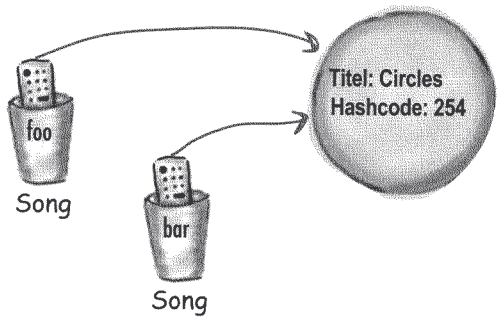
**Wenn zwei Objekte *foo* und *bar* gleich sind, muss *foo.equals(bar)* *true* liefern, und foo und bar müssen beim Aufruf von *hashCode()* den gleichen Wert zurückgeben. Damit ein Set zwei Objekte als gleich behandelt, müssen Sie die von der Klasse Object geerbten Methoden hashCode() und equals() überschreiben und so dafür sorgen, dass zwei verschiedene Objekte als gleich angesehen werden.**

## ➤ Referenzgleichheit

### Zwei Referenzen und ein Objekt auf dem Heap.

Zwei Referenzen, die auf dasselbe Objekt auf dem Heap verweisen, sind gleich. Punkt. Wenn Sie auf beiden Referenzen die Methode **hashCode()** aufrufen, erhalten Sie das gleiche Ergebnis. Wenn Sie die Methode hashCode() nicht überschreiben, zeigt sie ihr Standardverhalten (das sie, wie Sie sicher noch wissen, von der Klasse Object geerbt hat) – jedes Objekt erhält eine unverwechselbare Nummer. (Die meisten Java-Versionen weisen dem Objekt seinen Hashcode auf der Basis seiner Speicheradresse auf dem Heap zu, so dass zwei Objekte niemals den gleichen Hashcode haben.)

Wenn Sie wissen möchten, ob zwei *Referenzen* tatsächlich auf dasselbe Objekt verweisen, benutzen Sie den Operator ==, der (wie Sie wissen) die Bits in den Variablen vergleicht. Wenn beide Referenzen auf dasselbe Objekt zeigen, sind die Bits identisch.



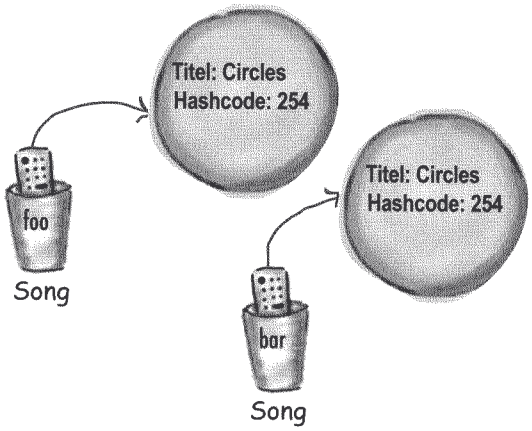

```
if (foo == bar) {
  // beide Referenzen verweisen auf
  // dasselbe Objekt auf dem Heap
}
```

## ➤ Objektgleichheit

### Zwei Referenzen und zwei Objekte auf dem Heap, aber die Objekte sind als *inhaltlich äquivalent* anzusehen.

Wenn Sie zwei verschiedene Song-Objekte als gleich behandeln wollen (wenn Sie beispielsweise beschließen würden, dass zwei Songs gleich sind, wenn ihre Titel-Variablen übereinstimmen), müssen Sie sowohl die Methode **hashCode()** als auch die Methode **equals()** überschreiben, die von der Klasse Object geerbt wurden.

Wenn Sie hashCode() *nicht* überschreiben, erhält jedes Objekt – wie oben bereits erwähnt – gemäß dem (von Object geerbten) Standardverhalten einen unverwechselbaren Hashcode-Wert. Sie müssen also hashCode() überschreiben, um sicherzugehen, dass zwei äquivalente Objekte tatsächlich den gleichen Hashcode zurückgeben. Und zudem müssen Sie equals() so überschreiben, dass Sie immer ***true*** erhalten, wenn Sie die Methode auf einem der beiden Objekte aufrufen und ihr dabei das jeweils andere Objekt übergeben.




```
if (foo.equals(bar) && foo.hashCode() == bar.hashCode()) {
  // beide Referenzen beziehen sich entweder auf
  // ein und dasselbe Objekt oder auf zwei als
  // gleich geltende Objekte
}
```

# Wie ein HashSet auf doppelt enthaltene Elemente prüft: hashCode() und equals()

Wenn Sie ein Objekt in ein HashSet setzen, entscheidet dieses anhand des Objekt-Hashcode-Werts, an welche Stelle es das Objekt innerhalb des Sets setzt. Es vergleicht aber auch den Hashcode des Objekts mit dem Hashcode aller anderen Objekte im HashSet, und wenn dort kein damit übereinstimmender Hashcode vorkommt, geht das HashSet davon aus, dass dieses neue Objekt keine Dublette ist.

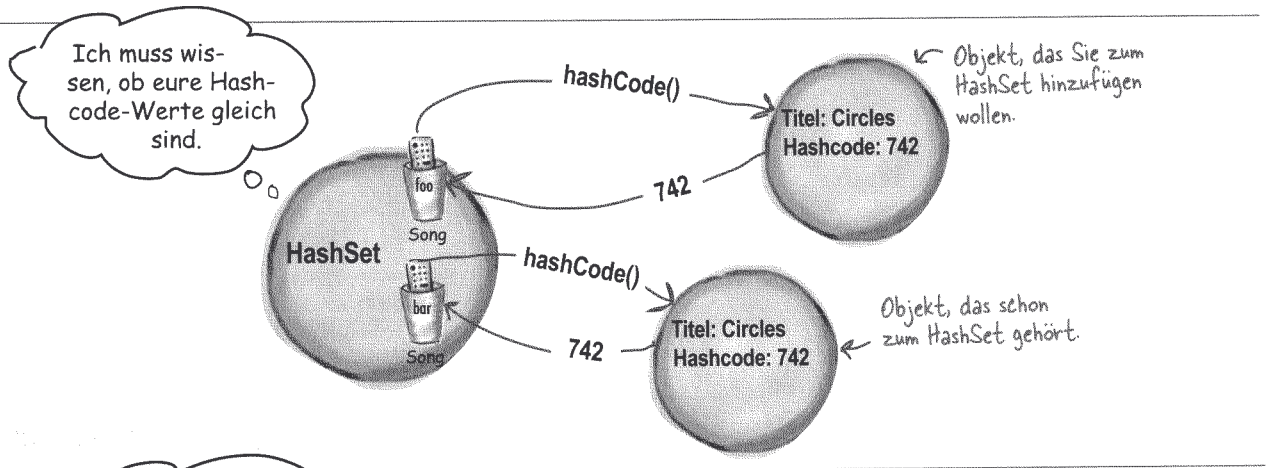
Mit anderen Worten: Wenn die Hashcodes unterschiedlich sind, geht das HashSet davon aus, dass die entsprechenden Objekte auf keinen Fall gleich sein können!

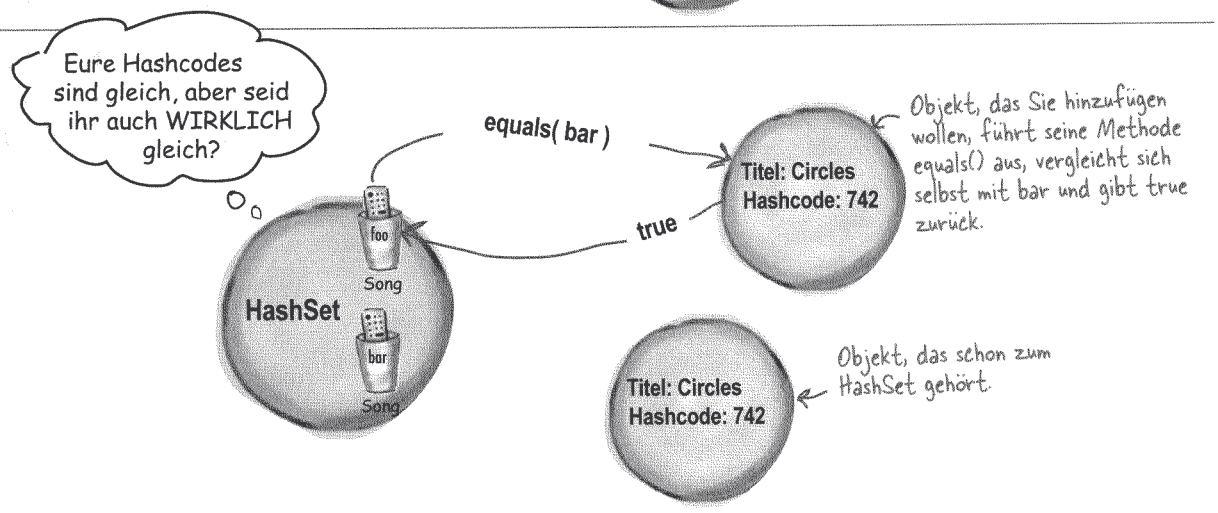
Sie müssen also hashCode() überschreiben, um dafür zu sorgen, dass die Objekte den gleichen Wert haben.

Aber auch zwei Objekte mit dem gleichen Hashcode sind möglicherweise *nicht gleich* (mehr dazu auf der nächsten Seite); wenn also das HashSet einen übereinstimmenden Hashcode bei zwei Objekten findet (bei dem einzufügenden Element und bei einem schon im Set enthaltenen Element), ruft es die equals()-Methode auf einem dieser Objekte auf, um festzustellen, ob diese Objekte mit dem übereinstimmenden Hashcode auch tatsächlich gleich *sind*.

Und wenn sie gleich sind, weiß das HashSet, dass das hinzuzufügende Objekt ein »Duplikat« von etwas im Set ist; daher fügt es das neue Objekt *nicht* hinzu.

Es wird keine Exception ausgelöst, aber die add()-Methode des HashSet gibt einen boolean-Wert zurück, um Ihnen (falls es Sie interessiert) mitzuteilen, ob das neue Objekt hinzugefügt wurde. Wenn Ihnen also die Methode add() *false* zurückgibt, wissen Sie, dass das neue Objekt ein »Duplikat« von irgendetwas war, was schon im Set vorkommt.



# Die Klasse Song mit den überschriebenen Methoden hashCode() und equals()

```
class Song implements Comparable<Song> {
  String titel;
  String künstler;
  String bewertung;
  String bpm;

  public boolean equals(Object einSong) {
    Song s = (Song) einSong;
    return getTitel().equals(s.getTitel());
  }

  public int hashCode() {
    return titel.hashCode();
  }

  public int compareTo(Song s) {
    return titel.compareTo(s.getTitel());
  }

  Song(String t, String k, String bw, String b) {
    titel = t;
    künstler = k;
    bewertung = bw;
    bpm = b;
  }

  public String getTitel() {
    return titel;
  }

  public String getKünstler() {
    return künstler;
  }

  public String getBewertung() {
    return bewertung;
  }

  public String getBpm() {
    return bpm;
  }

  public String toString() {
    return titel;
  }
}
```

Das HashSet (oder wer immer diese Methode aufruft) übergibt ihr einen anderen Song.

Eine GUTE Nachricht: Der Titel ist ein String, und Strings haben eine überschriebene equals()-Methode. Daher müssen wir hier nichts weiter tun, als den einen Titel zu fragen, ob er gleich dem anderen Songtitel ist.

Hier das gleiche Spiel ... Die Klasse String hat eine überschriebene hashCode()-Methode, daher können Sie einfach das Ergebnis eines Aufrufs von hashCode() auf dem Titel zurückgeben. Beachten Sie, dass hashCode() und equals() die GLEICHE Instanzvariable verwenden.

Es funktioniert! Keine doppelten Elemente mehr bei der Ausgabe des HashSet. Allerdings wurde die sortierte Reihenfolge nicht beibehalten, als wir die ArrayList in das HashSet gesetzt haben.

```
Datei Bearbeiten Fenster Hilfe DoppeltHältNichtBesser
% java Jukebox6

[Pink Moon, Somersault, Shiva Moon, Circles,
Deep Channel, Passenger, Listen, Listen,
Listen, Circles]

[Circles, Circles, Deep Channel, Listen,
Listen, Listen, Passenger, Pink Moon, Shiva
Moon, Somersault]

[Pink Moon, Listen, Shiva Moon, Circles,
Deep Channel, Passenger, Somersault]
```

## Das Java-Object-Gesetz für hashCode() und equals()

**In den API-Docs für die Klasse Object sind die Regeln genannt, die Sie befolgen MÜSSEN:**

- **Wenn zwei Objekte gleich sind, MÜSSEN sie übereinstimmende Hashcodes haben.**
- **Wenn zwei Objekte gleich sind, MUSS true zurückgegeben werden, wenn man equals() auf einem der beiden Objekte aufruft – egal auf welchem. Mit anderen Worten: wenn (a.equals(b)), dann (b.equals(a)).**
- **Wenn zwei Objekte den gleichen Hashcode-Wert haben, müssen sie NICHT unbedingt gleich sein. Aber WENN sie gleich sind, MÜSSEN sie den gleichen Hashcode-Wert haben.**
- **Wenn Sie equals() überschreiben, MÜSSEN Sie also auch hashcode() überschreiben.**
- **Das Standardverhalten von hashCode() ist, einen für jedes Objekt auf dem Heap unverwechselbaren ganzzahligen Wert zu erzeugen. Wenn also hashCode() in einer Klasse nicht überschrieben wird, können zwei verschiedene Objekte auf dem Heap NIEMALS als gleich angesehen werden.**
- **Standardmäßig führt equals() einen Vergleich mittels == durch. Anders ausgedrückt: Es wird geprüft, ob die beiden Referenzen auf ein und dasselbe Objekt auf dem Heap verweisen. Wenn also equals() in einer Klasse nicht überschrieben wird, können zwei verschiedene Objekte NIEMALS als gleich angesehen werden, da Referenzen auf zwei verschiedene Objekte immer ein unterschiedliches Bitmuster enthalten werden.**

***a.equals(b)* MUSS auch bedeuten, dass *a.hashCode() == b.hashCode()*.**

***Aber a.hashCode() == b.hashCode()* muss NICHT bedeuten, dass *a.equals(b)*.**

## Es gibt keine Dummen Fragen

**F: Wie kommt es, dass Objekte den gleichen Hashcode haben können, obwohl sie nicht gleich sind?**

A: HashSets verwenden Hashcodes, um die Elemente so zu speichern, dass viel schneller auf sie zugegriffen werden kann. Wenn Sie ein Objekt in einer ArrayList finden wollen, indem Sie der ArrayList eine Kopie des Objekts übergeben (statt eines Indexwerts), muss die ArrayList mit dem Suchen am Anfang beginnen und sich bei jedem Element in der Liste anschauen, ob es übereinstimmt. HashSet dagegen findet ein Objekt wesentlich schneller, da es den Hashcode als eine Art Etikett auf dem »Eimer« benutzt, in dem es das Element abgelegt hat. Wenn Sie also sagen: »Such mir ein Objekt in dem Set, das genau wie dieses hier ist ...«, holt sich das HashSet den Hashcode-Wert von der Kopie des Songs, die Sie ihm übergeben haben (sagen wir, 742), sagt dann: »Oh, ich weiß genau, wo das Objekt mit Hashcode Nr. 742 gespeichert ist ...«, und geht direkt zu dem Eimer mit der Nr. 742.

Mehr müssen Sie nicht wissen, um HashSets wirksam einzusetzen, obwohl man Ihnen in einem Hochschul-Informatikkurs noch viel mehr erzählen würde. Die Entwicklung eines guten Hashcode-Algorithmus ist tatsächlich das Thema so mancher Doktorarbeit – aber so eingehend wollen wir das hier lieber nicht besprechen.

Der springende Punkt ist, dass Objekte die gleichen Hashcodes haben können, ohne dass notwendigerweise auch die Objekte gleich sein müssen, weil der in der hashCode()-Methode verwendete »Hash-Algorithmus« zufällig den gleichen Wert für mehrere Objekte zurückgeben könnte. Und das heißt tatsächlich, dass mehrere Objekte alle im selben Eimer im HashSet landen würden (weil jeder Eimer genau einen Hashcode-Wert repräsentiert) – aber das ist nicht das Ende der Welt. Es könnte bedeuten, dass das HashSet einfach ein bisschen weniger effizient ist (oder mit einer extrem großen Zahl von Elementen gefüllt ist), aber wenn das HashSet mehr als ein Objekt im selben Hashcode-Eimer vorfindet, prüft es einfach mit Hilfe der equals()-Methode, ob eine perfekte Übereinstimmung vorliegt. Anders ausgedrückt: Hashcode-Werte werden manchmal benutzt, um die Suche weiter einzuengen, aber um die *eine* exakte Übereinstimmung zu finden, muss sich ein HashSet erst alle Objekte in diesem Eimer (dem für all die Objekte mit dem gleichen Hashcode) vornehmen und auf ihnen equals() aufrufen, um zu prüfen, ob das gesuchte Objekt dabei ist.

# Und wenn wir die Sortierung behalten wollen, nehmen wir TreeSet

Ähnlich wie HashSet verhindert auch TreeSet doppelt vorkommende Elemente. Aber zusätzlich hält es die Liste auch sortiert. Das funktioniert ähnlich wie bei der Methode sort(): Wenn Sie ein TreeSet mit Hilfe seines parameterlosen Konstruktors erzeugen, benutzt das TreeSet die compareTo()-Methode des jeweiligen Objekts für die Sortierung. Sie haben jedoch auch die Möglichkeit, stattdessen dem TreeSet-Konstruktor einen Comparator zu übergeben und es diesen verwenden zu lassen. Der Nachteil an TreeSet ist: Wenn Sie keine Sortierung *benötigen*, müssen Sie trotzdem eine kleine Leistungsbeeinträchtigung in Kauf nehmen. Aber wahrscheinlich werden Sie feststellen, dass diese Beeinträchtigung bei den meisten Anwendungen kaum zu bemerken ist.

```
import java.util.*;
import java.io.*;
public class Jukebox8 {
  ArrayList<Song> songList = new ArrayList<Song>();

  public static void main(String[] args) {
    new Jukebox8().los();
  }

  public void los() {
    getSongs();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);
    TreeSet<Song> songSet = new TreeSet<Song>();
    songSet.addAll(songList);
    System.out.println(songSet);
  }

  void getSongs() {
    try {
      File datei = new File("SongListPlus.txt");
      BufferedReader reader = new BufferedReader(new FileReader(datei));
      String zeile = null;
      while ((zeile= reader.readLine()) != null) {
        addSong(zeile);
      }

    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  void addSong(String zuParsendeZeile) {
    String[] tokens = zuParsendeZeile.split("/");
    Song nächsterSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
    songList.add(nächsterSong);
  }
}
```

Es wird ein TreeSet statt eines HashSet instantiiert. Der Aufruf des parameterlosen TreeSet-Konstruktors bedeutet, dass das Set die compareTo()-Methode des Songs zum Sortieren benutzen wird.

(Wir hätten auch einen Comparator übergeben können.)

Mit addAll() können wir alle Songs aus dem HashSet auf einmal hinzufügen. (Wir hätten die Songs auch einzeln mit songSet.add() hineinsetzen können, wie wir es beim Hinzufügen der Songs zur ArrayList gemacht haben.)

# Was Sie über TreeSet wissen MÜSSEN ...

TreeSet sieht einfach aus, aber Sie sollten sich unbedingt darüber im Klaren sein, welche Voraussetzungen bei seiner Verwendung erfüllt sein müssen. Wir halten das für so wichtig, dass wir eine Übung daraus gemacht haben, damit Sie darüber nachdenken *müssen. Das meinen wir ernst.*

Sehen Sie sich diesen Code an. Lesen Sie ihn sorgfältig und beantworten Sie dann die unten stehenden Fragen. (Anmerkung: Dieser Code enthält keine Syntaxfehler.)

```
import java.util.*;

public class TestTree {
  public static void main (String[] args) {
    new TestTree().los();
  }

  public void los() {
    Buch b1 = new Buch("Katzen sind unvergleichlich");
    Buch b2 = new Buch("Sortieren Sie Ihren Körper neu");
    Buch b3 = new Buch("Findet Emo");

    TreeSet<Buch> tree = new TreeSet<Buch>();
    tree.add(b1);
    tree.add(b2);
    tree.add(b3);
    System.out.println(tree);
  }
}

class Buch {
  String titel;
  public Buch(String t) {
    titel = t;
  }
  // überschriebene toString()-Methode
}
```

1.) Was passiert, wenn Sie diesen Code kompilieren?

2.) Falls er kompilierbar ist: Welches Ergebnis erhält man bei der Ausführung der Klasse TestTree?

3.) Falls es ein Problem mit diesem Code gibt (entweder zur Kompilierzeit oder zur Laufzeit): Wie könnten Sie es in Ordnung bringen?

# TreeSet-Elemente MÜSSEN vergleichbar sein

Das TreeSet kann nicht die Gedanken des Programmierers lesen, um herauszufinden, wie die Objekte sortiert werden sollen. Sie müssen ihm schon sagen, *wie*.

## Wenn Sie ein TreeSet verwenden wollen, muss eine von diesen Bedingungen zutreffen:

### ➤ Die Elemente in der Liste müssen von einem Typ sein, der *Comparable* implementiert.

Die Klasse Buch auf der letzten Seite implementiert Comparable nicht, daher würde sie zur Laufzeit nicht funktionieren. Überlegen Sie mal: Der einzige Lebenszweck des armen TreeSet ist es doch, Ihre Elemente sortiert zu halten – und wieder mal hatte es keine Ahnung, wie Buch-Objekte zu sortieren sind! Zur Kompilierzeit ist alles in Ordnung, weil die add()-Methode des TreeSet kein Argument vom Typ Comparable erwartet. Die add()-Methode von TreeSet nimmt alles an, was dem Typ entspricht, den Sie bei der Erzeugung des TreeSet verwendet haben. Mit anderen Worten: Wenn Sie new TreeSet<Buch>() schreiben, entspricht die add()-Methode praktisch add(Book). Und es wird nicht vorausgesetzt, dass die Klasse Buch Comparable implementiert! Aber zur Laufzeit geht es schief, wenn Sie das zweite Element zum Set hinzufügen. Hier versucht das Set zum ersten Mal, die compareTo()-Methode eines der Objekte aufzurufen, und ... es geht nicht.

```
class Buch implements Comparable {
  String titel;
  public Buch(String t) {
    titel = t;
  }
  public int compareTo(Object b) {
    Buch buch = (Buch) b;
    return (titel.compareTo(buch.titel));
  }
  // überschriebene toString()-Methode
}
```

# ODER

### ➤ Sie verwenden den überladenen Konstruktor von TreeSet, dem ein *Comparator* übergeben wird.

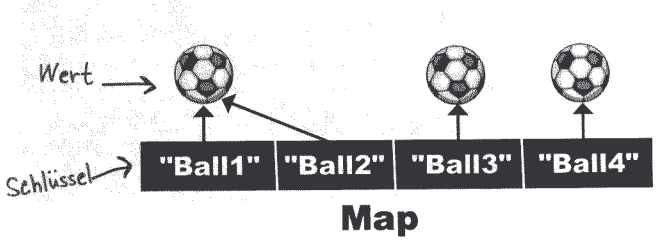
TreeSet funktioniert in vieler Hinsicht wie die Methode sort(): Sie können wählen, ob Sie die compareTo()-Methode des Elements benutzen wollen – vorausgesetzt, die Klasse des Elements implementiert das Interface Comparable – ODER ob Sie einen selbst geschriebenen Comparator verwenden wollen, der weiß, wie man die Elemente in dem Set sortiert. Wenn Sie einen eigenen Comparator benutzen wollen, rufen Sie den TreeSet-Konstruktor auf, der einen Comparator entgegennimmt.

```
public class BuchVergleich implements Comparator<Buch> {
  public int compare(Buch eins, Buch zwei) {
    return (eins.titel.compareTo(zwei.titel));
  }
}

public class TestTree {
  // hier steht die main-Methode
    public void los() {
    Buch b1 = new Buch("Katzen sind unvergleichlich");
    Buch b2 = new Buch("Sortieren Sie Ihren Körper neu");
    Buch b3 = new Buch("Findet Emo");
    BuchVergleich bVergleich = new BuchVergleich();
    TreeSet<Buch> tree = new TreeSet<Buch>(bVergleich);
    tree.add(b1);
    tree.add(b2);
    tree.add(b3);
    System.out.println(tree);
  }
}
```

# Lists und Sets kennen wir – jetzt nehmen wir eine Map

Lists und Sets sind prima, aber manchmal ist die Collection der Wahl eine Map. Erinnern Sie sich: Maps gehören zu den Java-Collections (im weiteren Sinn), implementieren aber nicht das Interface Collection.

Stellen Sie sich vor, Sie brauchen eine Collection, die wie eine Eigenschaftsliste funktioniert – wenn Sie ihr einen Namen übergeben, gibt sie Ihnen den damit verknüpften Wert zurück. Die Schlüssel sind zwar oft Strings, aber eigentlich kann jedes Java-Objekt (oder – durch Autoboxing – jeder primitive Wert) als Schlüssel fungieren.




**Jedes Element einer Map besteht eigentlich aus ZWEI Objekten: einem *Schlüssel* und einem *Wert*. Werte können mehrfach vorkommen, Schlüssel dagegen nicht.**

## Beispiel für eine Map

```
import java.util.*;

public class TestMap {

  public static void main(String[] args) {

    HashMap<String, Integer> punktzahlen = new HashMap<String, Integer>();

    punktzahlen.put("Melanie", 42);
    punktzahlen.put("Max", 343);
    punktzahlen.put("Miriam", 420);

    System.out.println(punktzahlen);
    System.out.println(punktzahlen.get("Max"));
  }
}
```

HashMap braucht ZWEI Typparameter – einen für den Schlüssel und einen für den Wert.

Verwenden Sie put() statt add(); und jetzt müssen Sie natürlich zwei Argumente (Schlüssel, Wert) übergeben.

Die Methode get() nimmt einen Schlüssel entgegen und gibt den Wert zurück (in diesem Fall ein Integer).

```
Datei Bearbeiten Fenster Hilfe WoBinIch
%java TestMap

{Miriam=420, Max=343, Melanie=42}
343
```

Wenn Sie eine Map ausgeben, erhalten Sie Schlüssel=Wert-Paare, und zwar in geschweiften Klammern {} statt der eckigen[] wie bei Lists und Sets.

# Endlich zurück zu Generics

Erinnern Sie sich: Weiter vorn in diesem Kapitel haben wir darüber gesprochen, dass Methoden, die Argumente mit generischen Typen annehmen, irgendwie ... *komisch* sein können. Wir meinen komisch im Hinblick auf Polymorphie. Wenn es hier anfängt, Ihnen merkwürdig vorzukommen, lesen Sie einfach weiter – es dauert ein paar Seiten, bis die ganze Geschichte erzählt ist.

Als Erstes wollen wir uns ins Gedächtnis zurückrufen, wie Polymorphie bei *Array*-Argumenten funktioniert, und dann sehen wir uns dasselbe bei einer generischen List an. Der folgende Code lässt sich ohne Fehlermeldungen kompilieren und ausführen:

**Wenn das Argument einer Methode ein *Array* von Haustieren ist, kann man der Methode auch ein Array eines beliebigen Haustier-Untertyps übergeben.**

**Mit anderen Worten, wenn eine Methode wie folgt deklariert wird:**

***void foo(Haustier[] t){}***

**Angenommen, Hund erweitert Haustier, dann sind beide Aufrufe möglich:**

***foo(einHaustierArray);***
***foo(einHundArray);***

## So funktioniert es mit normalen *Arrays:*

```
import java.util.*;

public class TestGenerics1 {
  public static void main(String[] args) {
    new TestGenerics1().los();
  }

  public void los() {
    Haustier[] haustiere = {new Hund(), new Katze(), new Hund()};
    Hund[] hunde = {new Hund(), new Hund(), new Hund()};
    tiereAnnehmen(haustiere);
    tiereAnnehmen(hunde);
  }

  public void tiereAnnehmen(Haustier[] haustiere) {
    for(Haustier h: haustiere) {
      h.fressen();
    }
  }
}
```

Deklarieren und erzeugen Sie ein Haustier-Array, das sowohl Hunde als auch Katzen enthält.

Deklarieren und erzeugen Sie ein Hund-Array, das nur Hunde enthält (der Compiler lässt Sie dort keine Katze hineinsperren).

Rufen Sie tiereAnnehmen() auf, mit beiden Array-Typen als Argument ...

Der springende Punkt ist, dass die Methode tiereAnnehmen() sowohl ein Haustier[] als auch ein Hund[] annimmt, denn Hund IST-EIN Haustier. Polymorphie >>in Aktion<<.

Erinnern Sie sich, wir können NUR die Methoden aufrufen, die im Typ Haustier deklariert sind, da der Parameter haustiere vom Typ Haustier-Array ist und wir keinen Cast durchgeführt haben. (In welchen Typ sollten wir das Haustier auch umwandeln? Das Array kann ja sowohl Hunde als auch Katzen enthalten.)

```
abstract class Haustier {
  void fressen() {
    System.out.println("Haustier frisst");
  }
}
class Hund extends Haustier {
  void bellen() { }
}
class Katze extends Haustier {
  void miauen() { }
}
```

Die vereinfachte Haustier-Klassenhierarchie.

# Polymorphe Argumente mit Generics

Wir haben also gesehen, wie das Ganze mit Arrays funktioniert, aber funktioniert es auch genau so, wenn wir von einem Array auf eine ArrayList umsteigen? Hört sich eigentlich vernünftig an, oder?

Versuchen wir es zunächst nur mit der Haustier-ArrayList. Wir haben lediglich die Methode los() ein wenig verändert:

### Nur ArrayList<Haustier> als Argument:

Ein einfacher Austausch von Haustier[] gegen ArrayList<Haustier>.

```
public void los() {
  ArrayList<Haustier> haustiere = new ArrayList<Haustier>();
  haustiere.add(new Hund());
  haustiere.add(new Katze());
  haustiere.add(new Hund());

  tiereAnnehmen(haustiere);
}
```

← Wir müssen eines nach dem anderen hinzufügen, weil es hier – anders als bei der Array-Erzeugung – keine abkürzende Syntax gibt.

← Dies ist der gleiche Code, abgesehen davon, dass die Variable »haustiere« jetzt eine ArrayList referenziert statt ein Array

```
public void tiereAnnehmen(ArrayList<Haustier> haustiere) {
  for(Haustier h: haustiere) {
    h.fressen();
  }
}
```

Die Methode erhält nun eine ArrayList statt eines Arrays, aber alles andere ist gleich. Erinnern Sie sich, diese for-Schleifen-Syntax funktioniert sowohl bei Arrays als auch bei Collections.

### Kompiliert und läuft korrekt

```
Datei Bearbeiten Fenster Hilfe KatzenfutterSchmecktBesser
% java TestGenerics2

Haustier frisst
Haustier frisst
Haustier frisst
```

# Ob es wohl auch mit ArrayList<Hund> klappt?

Auf Grund der Polymorphie gestattet uns der Compiler, einer Methode mit einem Haustier-Array-Argument ein Hund-Array zu übergeben. Kein Problem. Und eine ArrayList<Haustier> kann einer Methode mit einem ArrayList<Haustier>-Argument übergeben werden. Die große Frage lautet: Wird das ArrayList<Haustier>-Argument auch eine ArrayList<Hund> akzeptieren? Bei Arrays funktioniert es – sollte es hier nicht auch funktionieren?

## Übergabe der ArrayList<Hund>:

```
public void los() {
  ArrayList<Haustier> haustiere = new ArrayList<Haustier>();
  haustiere.add(new Hund());
  haustiere.add(new Katze());
  haustiere.add(new Hund());
  tiereAnnehmen(haustiere);

  ArrayList<Hund> hunde = new ArrayList<Hund>();
  hunde.add(new Hund());
  hunde.add(new Hund());
  tiereAnnehmen(hunde);
}

public void tiereAnnehmen(ArrayList<Haustier> haustiere) {
  for(Haustier h: haustiere) {
    h.fressen();
  }
}
```

← Wir wissen, dass diese Zeile prima funktioniert hat.

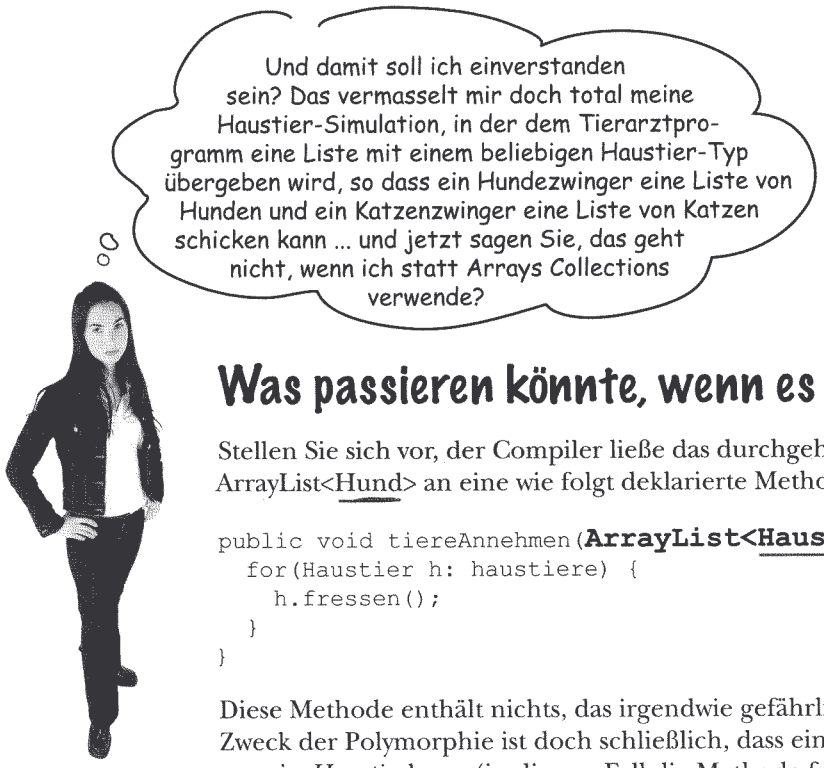
Wir machen eine Hund-ArrayList und stecken ein paar Hunde hinein.

← Wird dies noch funktionieren, nachdem wir von einem Array auf eine ArrayList umgestiegen sind?

## Nach der Kompilierung:

```
Datei Bearbeiten Fenster Hilfe KatzenSindKlüger
% javac TestGenerics3.java

TestGenerics3.java:21: tiereAnnehmen(java.util.
ArrayList<Haustier>) in TestGenerics3 cannot be applied to
(java.util.ArrayList<Hund>)
      tiereAnnehmen(hunde);
      ^
1 error
```

Es sah so richtig aus, und doch ist was falsch gelaufen ...




# Was passieren könnte, wenn es erlaubt wäre ...

Stellen Sie sich vor, der Compiler ließe das durchgehen. Dann könnten Sie eine ArrayList<Hund> an eine wie folgt deklarierte Methode übergeben:

```
public void tiereAnnehmen(ArrayList<Haustier> haustiere) {
  for(Haustier h: haustiere) {
    h.fressen();
  }
}
```

Diese Methode enthält nichts, das irgendwie gefährlich *aussieht*, oder? Sinn und Zweck der Polymorphie ist doch schließlich, dass ein Hund all das tun kann, was ein *Haustier* kann (in diesem Fall die Methode fressen()). Wo liegt also das Problem bei einem Aufruf der Methode fressen() auf den einzelnen Hund-Referenzen?

*Nirgendwo.* Es gibt kein Problem.

Mit *dem* Code ist alles in Ordnung. Aber nehmen Sie stattdessen *diesen* Code:

```
public void tiereAnnehmen(ArrayList<Haustier> haustiere) {
  haustiere.add(new Katze());
}
```

← Oh Schreck!! Gerade haben wir eine Katze in etwas hineingesteckt, was eine Nur-Hund-ArrayList sein könnte.

Also da liegt das Problem. Es ist bestimmt nichts Verkehrtes daran, eine Katze in eine ArrayList<Haustier> zu stecken; das ist ja der Sinn einer ArrayList mit einem Supertyp wie z.B. Haustier – dass Sie alle Haustier-Typen in eine einzige Haustier-ArrayList stecken können.

Aber wenn Sie dieser Methode, die eine Haustier-ArrayList entgegennimmt, eine Hund-ArrayList übergeben würden, d.h. eine ArrayList, die eigentlich ausschließlich Hunde enthalten sollte – dann hätten Sie plötzlich eine Katze in der Hund-Liste. Der Compiler weiß: Wenn er erlaubt, dass einer solchen Methode eine Hund-ArrayList übergeben wird, könnte irgendjemand zur Laufzeit eine Katze zu der Hundeliste hinzufügen. Und dieses Risiko lässt der Compiler Sie gar nicht erst eingehen.

***Wenn Sie eine Methode mit dem Argument ArrayList<Haustier> deklarieren, können Sie ihr NUR eine ArrayList<Haustier> übergeben, keine ArrayList<Hund> oder ArrayList<Katze>.***

Moment mal ... wenn Sie aus diesem Grund einer Methode, die eine Haustier-ArrayList erwartet, keine Hund-ArrayList übergeben dürfen - damit Sie nicht vielleicht eine Katze in etwas hineinstecken, was eigentlich eine Hund-List ist -, warum funktioniert es dann bei Arrays? Haben Sie bei Arrays nicht das gleiche Problem? Könnten Sie nicht auch ein Katzenobjekt zu einem Hund[] hinzufügen?

## Array-Typen werden zur Laufzeit noch einmal geprüft, aber Collection-Typen nur beim Kompilieren

Nehmen wir an, Sie fügen eine Katze zu einem Array hinzu, das als Hund[] deklariert ist (ein Array, das einem als Haustier[] deklarierten Methodenargument übergeben wurde, was eine bei Arrays absolut zulässige Zuweisung ist).

```
public void los() {
  Hund[] hunde = {new Hund(), new Hund(), new Hund()};
  tiereAnnehmen(hunde);
}

public void tiereAnnehmen(Haustier[] haustiere) {
  haustiere[0] = new Katze();
}
```

Wir haben eine Katze in ein Hund-Array gesteckt. Der Compiler hat es erlaubt, weil er weiß, dass Sie der Methode ein Katze-Array oder ein Haustier-Array hätten übergeben können – also hielt der Compiler es für möglich, dass das in Ordnung war.

### Es lässt sich kompilieren, aber bei der Ausführung:

Puh! Wenigstens hat die JVM es verhindert.

```
Datei Bearbeiten Fenster Hilfe KatzenSindKlüger
% java TestGenerics1
Exception in thread "main" java.lang.ArrayStoreException:
Katze
        at TestGenerics1.tiereAnnehmen(TestGenerics1.java:16)
        at TestGenerics1.los(TestGenerics1.java:12)
        at TestGenerics1.main(TestGenerics1.java:5)
```

Wäre es nicht wundervoll, wenn es eine Möglichkeit gäbe, trotzdem polymorphe Collection-Typen als Methodenargumente zu verwenden, so dass mein Tierarztprogramm Hund-Listen und Katze-Listen annehmen könnte? So könnte ich die Listen mit einer Schleife durchlaufen und die Methode immunisieren() aufrufen – aber trotzdem müsste es noch sicher sein, damit man nicht eine Katze in eine Hund-Liste stecken kann. Aber das ist wohl nur ein Traum ...

# Wildcards (Jokerzeichen) kommen zu Hilfe

Es sieht zwar ungewöhnlich aus, aber es gibt tatsächlich einen Weg, ein Methodenargument zu erzeugen, das eine ArrayList eines beliebigen Haustier-Untertyps aufnehmen kann. Der einfachste Weg ist, ein **Wildcard** (einen Platzhalter; der englische Begriff Wildcard bezeichnet ursprünglich den Joker in einem Kartenspiel) zu verwenden. Wildcards wurden speziell aus diesem Grund in die Sprache Java aufgenommen.

```
public void tiereAnnehmen(ArrayList<? extends Haustier> haustiere) {
  for(Haustier h: haustiere) {
    h.fressen();
  }
}
```

Erinnern Sie sich, das Schlüsselwort »extends« bedeutet hier je nach Typ entweder extends ODER implements. Wenn Sie also eine ArrayList von Typen entgegennehmen möchten, die das Interface Haustier implementieren, deklarieren Sie sie wie folgt:

ArrayList<? extends Haustier>

Jetzt fragen Sie sich: »Wo liegt der *Unterschied?* Hat man nicht das gleiche Problem wie vorher? Die obige Methode macht ja nichts Gefährliches – sie ruft nur eine Methode auf, die jeder Untertyp von Haustier garantiert haben muss. Aber könnte nicht immer noch jemand das so ändern, dass eine Katze zur Liste *haustiere* hinzugefügt wird, obwohl diese in Wirklichkeit eine ArrayList<Hund> ist? Und das wird ja zur Laufzeit nicht erneut geprüft – also worin unterscheidet es sich dann überhaupt von einer Deklaration ohne Jokerzeichen?«

Und Sie haben Recht, sich das zu fragen. Die Antwort ist NEIN. Wenn Sie in Ihrer Deklaration ein Wildcard <?> verwenden, hindert der Compiler Sie daran, irgendetwas zu tun, durch das etwas zur Liste hinzugefügt würde!

**Wenn Sie in Ihrem Methodenargument ein Jokerzeichen verwenden, HINDERT der Compiler Sie an allem, was die in dem Methodenparameter referenzierte Liste beschädigen könnte.**

**Sie können zwar noch Methoden auf den Elementen in der Liste aufrufen, aber nichts zur Liste hinzufügen.**

**Anders ausgedrückt: Sie können etwas mit den Listenelementen machen, aber nichts Neues in die Liste hineinstecken. So sind Sie zur Laufzeit vor bösen Überraschungen sicher, denn der Compiler lässt Sie nichts tun, was zur Laufzeit schlimme Folgen haben könnte.**

**Das hier ist innerhalb von tiereAnnehmen() okay:**

```
for(Haustier h: haustiere) {
  h.fressen();
}
```

**Aber DIES hier ist nicht kompilierbar:**

```
haustiere.add(new Katze());
```

# Das Gleiche mit einer anderen Syntax

Sie erinnern sich wahrscheinlich, dass wir in den Javadocs der Methode sort() gesehen haben, wie ein generischer Typ in einem ungewöhnlichen Format verwendet wurde – der Typparameter wurde vor dem Rückgabetyp deklariert. Das ist nur eine andere Art und Weise, den Typparameter zu deklarieren, aber das Ergebnis ist das gleiche:

### Dies hier:

```
public <T extends Haustier> void etwasAufnehmen(ArrayList<T> list)
```

### ... macht das Gleiche wie dies:

```
public void etwasAufnehmen(ArrayList<? extends Haustier> list)
```

**F: Wenn beides das Gleiche macht – gibt es Gründe, eine Version zu bevorzugen?**

A: Das hängt davon ab, ob Sie »T« noch an irgendeiner anderen Stelle verwenden wollen. Was ist beispielsweise, wenn Ihre Methode *zwei* Argumente und vielleicht noch einen Rückgabewert haben soll – alles Listen mit demselben Typ, der Haustier erweitert? In diesem Fall dürfen Sie den Typparameter nur einmal deklarieren, um nicht verschiedene Typen zu bekommen, die Haustier erweitern:

```
public <T extends Haustier> ArrayList<T> mischen(
       ArrayList<T> eins, ArrayList<T> zwei)
```

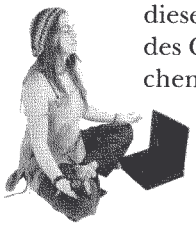
Hier stellt der Compiler sicher, dass T in allen drei Fällen derselbe Typ ist.

Wenn Sie das NICHT einschränken wollen, können Sie natürlich auch die Wildcard-Variante verwenden:

```
public ArrayList<? extends Haustier> mischen(
                ArrayList<? extends Haustier> eins,
                ArrayList<? extends Haustier> zwei)
```

Aber wollen Sie das wirklich riskieren? Können Sie sich vorstellen, was passiert, wenn man Hunde und Katzen zusammen in einen Behälter für Goldfische steckt? Also wir nicht ...

# Spielen Sie Compiler (für Fortgeschrittene)



Ihr Job ist es, Compiler zu spielen und festzustellen, welche dieser Anweisungen sich kompilieren lassen würden. Ein Teil des Codes wurde allerdings in diesem Kapitel nicht besprochen, daher müssen Sie die Antworten basierend auf dem hier Gelernten herausfinden, indem Sie die Regeln auf die neuen Situationen anwenden. In manchen Fällen müssen Sie vielleicht raten, aber das ist genau der Sinn der Sache: dass Sie sich anhand dessen, was Sie schon wissen, eine sinnvolle Antwort überlegen.

(Anmerkung: Nehmen Sie an, dieser Code stünde innerhalb einer zulässigen Klasse und Methode.)

**Kompilierbar?**

- ☐ `ArrayList<Hund> hunde1 = new ArrayList<Haustier>();`
- ☐ `ArrayList<Haustier> haustiere1 = new ArrayList<Hund>();`
- ☐ `List<Haustier> list = new ArrayList<Haustier>();`
- ☐ `ArrayList<Hund> hunde = new ArrayList<Hund>();`
- ☐ `ArrayList<Haustier> haustiere = hunde;`
- ☐ `List<Hund> hundList = hunde;`
- ☐ `ArrayList<Object> objects = new ArrayList<Object>();`
- ☐ `List<Object> objList = objects;`
- ☐ `ArrayList<Object> objs = new ArrayList<Hund>();`

# Lösungen zur »Reverse-Engineering-Übung«

```
import java.util.*;
public class BergSortierer {
  LinkedList<Berg> berge = new LinkedList<Berg>();
  class Namensvergleich implements Comparator <Berg> {
    public int compare(Berg eins, Berg zwei) {
      return eins.name.compareTo(zwei.name);
    }
  }
  class Höhenvergleich implements Comparator <Berg> {
    public int compare(Berg eins, Berg zwei) {
      return (zwei.höhe - eins.höhe);
    }
  }
  public static void main(String [] args) {
    new BergSortierer().los();
  }
  public void los() {
    berge.add(new Berg("Matterhorn", 4478));
    berge.add(new Berg("Dom", 4545));
    berge.add(new Berg("Nordend", 4609));
    berge.add(new Berg("Nadelhorn", 4327));
    System.out.println("Wie eingegeben:\n" + berge);
    Namensvergleich nc = new Namensvergleich();
    Collections.sort(berge, nc);
    System.out.println("Nach Namen:\n" + berge);
    Höhenvergleich hc = new Höhenvergleich();
    Collections.sort(berge, hc);
    System.out.println("Nach Höhe:\n" + berge);
  }
}
class Berg {
  String name;
  int höhe;

  Berg(String n, int h) {
    name = n;
    höhe = h;
  }
  public String toString( ) {
    return name + " " + höhe;
  }
}
```

Haben Sie die ABSTEIGENDE Reihenfolge in der Höhenliste bemerkt? :)

**Ausgabe:**

```
Datei Bearbeiten Fenster Hilfe BergeVersetzen
% java BergSortierer
Wie eingegeben:
[Matterhorn 4478, Dom 4545, Nordend 4609, Nadelhorn 4327]
Nach Namen:
[Dom 4545, Matterhorn 4478, Nadelhorn 4327, Nordend 4609]
Nach Höhe:
[Nordend 4609, Dom 4545, Matterhorn 4478, Nadelhorn 4327]
```

# Lösung der Übung

**Mögliche Antworten:**

Comparator

Comparable

compareTo( )

compare( )

ja

nein

Gegeben ist folgende kompilierbare Anweisung:

```
Collections.sort(meineArrayList);
```

1. Was muss die Klasse der in `meineArrayList` gespeicherten Objekte implementieren? **Comparable**
2. Welche Methode muss die Klasse der in `meineArrayList` gespeicherten Objekte implementieren? **compareTo()**
3. Kann die Klasse der in `meineArrayList` gespeicherten Objekte sowohl Comparator als auch Comparable implementieren? **ja**

Gegeben ist folgende kompilierbare Anweisung:

```
Collections.sort(meineArrayList, meinVergleich);
```

4. Kann die Klasse der in `meineArrayList` gespeicherten Objekte `Comparable` implementieren? **ja**
5. Kann die Klasse der in `meineArrayList` gespeicherten Objekte `Comparator` implementieren? **ja**
6. Muss die Klasse der in `meineArrayList` gespeicherten Objekte `Comparable` implementieren? **nein**
7. Muss die Klasse der in `meineArrayList` gespeicherten Objekte `Comparator` implementieren? **nein**
8. Was muss die Klasse des Objekts `meinVergleich` implementieren? **Comparator**
9. Welche Methode muss die Klasse des Objekts `meinVergleich` implementieren? **compare()**

# Spielen Sie Compiler - Lösung

**Kompilierbar?**

- [ ] `ArrayList<Hund> hunde1 = new ArrayList<Haustier>();`
- [ ] `ArrayList<Haustier> haustiere1 = new ArrayList<Hund>();`
- [x] `List<Haustier> list = new ArrayList<Haustier>();`
- [x] `ArrayList<Hund> hunde = new ArrayList<Hund>();`
- [ ] `ArrayList<Haustier> haustiere = hunde;`
- [x] `List<Hund> hundList = hunde;`
- [x] `ArrayList<Object> objects = new ArrayList<Object>();`
- [x] `List<Object> objList = objects;`
- [ ] `ArrayList<Object> objs = new ArrayList<Hund>();`

# Veröffentlichen Sie Ihren Code

**Es ist Zeit loszulassen.** Sie haben Ihren Code geschrieben. Sie haben ihn getestet. Sie haben ihn verbessert. Sie haben allen Bekannten erzählt, es wäre schön, wenn Sie nie wieder eine einzige Zeile Code sehen würden. Und doch haben Sie zu guter Letzt ein Kunstwerk geschaffen. Tatsächlich, das Ding läuft! Aber was nun? *Wie* bringen Sie es an die Frau oder den Mann? *Was* genau geben Sie den Endanwendern? Was ist, wenn Sie noch nicht einmal wissen, wer Ihre Endanwender sind? In diesen letzten beiden Kapiteln befassen wir uns damit, wie Sie Ihren Java-Code organisieren, packen und zum Einsatz bringen (was man auch als »Deployment« bezeichnet). Wir sehen uns die Möglichkeiten für lokales, halblokales und Ferndeployment an; dazu gehören ausführbare JAR-Dateien, Java Web Start, RMI und Servlets. In diesem Kapitel verbringen wir die meiste Zeit mit dem Organisieren und Packen des Codes – etwas, das Sie auf jeden Fall können müssen, unabhängig davon, welche Art von Deployment Sie letztendlich wählen. Zum Abschluss des Buchs befassen wir uns dann im letzten Kapitel mit einer der tollsten Möglichkeiten, die Java überhaupt bietet. Entspannen Sie sich. Wenn Sie Ihren Code zur Veröffentlichung freigeben, ist das kein Abschied für immer. Es gibt ja noch die Wartung ...

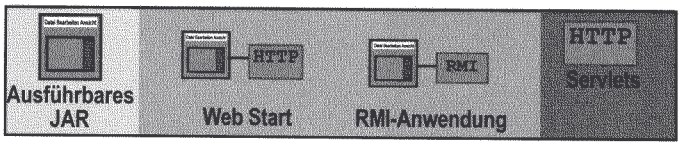
# Das Deployment Ihrer Anwendung

Was genau *ist* eine Java-Anwendung? Mit anderen Worten: Wenn Sie mit der Entwicklung fertig sind, was liefern Sie dann aus? Höchstwahrscheinlich werden Ihre Endanwender nicht genau das gleiche System haben wie Sie. Und vor allem: Die Endanwender haben auch Ihr Programm noch nicht! Es wird also Zeit, Ihr Programm für den Einsatz in der Welt da draußen – das Deployment – in Form zu bringen. In diesem Kapitel befassen wir uns mit lokalem Deployment (dazu gehören z.B. ausführbare JAR-Dateien) und mit einer als Java Web Start bezeichneten Technik, die aus lokalen und aus Fernanteilen besteht. Im nächsten Kapitel sehen wir uns dann die Deployment-Möglichkeiten mit noch größerem Fernanteil an; dazu gehören unter anderem RMI und Servlets.

Ein Java-Programm ist ein Bündel von Klassen. Es ist das Ergebnis Ihrer Entwicklungsarbeit.

Die eigentliche Frage lautet: Was machen Sie mit diesen Klassen, wenn Sie fertig sind?

**Deployment-Optionen**




100% lokal gemischt 100% fern

1. **Lokales Deployment**
   Die gesamte Anwendung läuft auf dem Computer des Endbenutzers als Stand-alone-Programm (wahrscheinlich über eine Benutzeroberfläche); das Deployment erfolgt mit Hilfe eines ausführbaren JAR (wir sehen uns JAR ein paar Seiten weiter hinten an).

2. **Lokal- und Ferndeployment gemischt**
   Die Anwendung wird über einen Client-Anteil ausgeliefert, der auf dem lokalen System des Benutzers läuft. Er ist mit einem Server verbunden, auf dem andere Teile der Anwendung laufen.

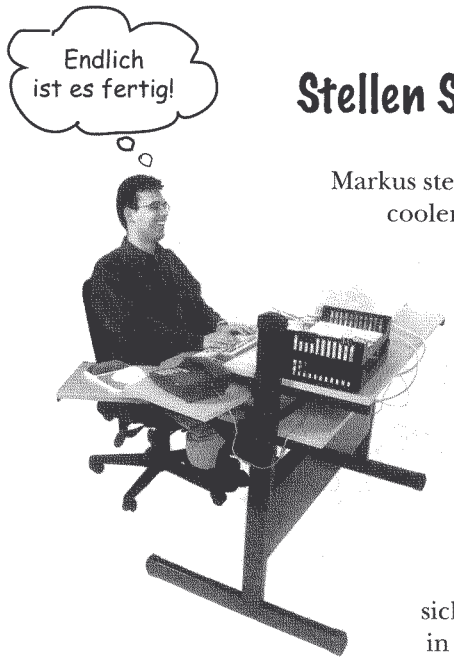
3. **Ferndeployment**
   Die gesamte Java-Anwendung läuft auf einem Server-System, und der Client greift über Nicht-Java-Anwendungen - wahrscheinlich über einen Webbrowser - auf dieses System zu.

Doch bevor wir uns richtig in diese ganze Deployment-Geschichte hineinknien, gehen wir einen Schritt zurück und sehen uns an, was passiert, wenn Sie Ihre Anwendung fertig programmiert haben und einfach die class-Dateien herausholen möchten, um sie an den Endanwender weiterzugeben. Was liegt eigentlich alles in diesem Arbeitsverzeichnis?

## Gewichtheben fürs Gehirn

Welche Vor- und Nachteile hat es, wenn Sie Ihr Java-Programm als lokale Stand-alone-Anwendung ausliefern, die auf dem Computer des Endanwenders läuft?

Welche Vor- und Nachteile hat es, wenn Sie Ihr Java-Programm als webbasiertes System ausliefern, bei dem der Anwender einen Webbrowser benutzt und der Java-Code in Form von Servlets auf dem Server läuft?




## Stellen Sie sich dieses Szenario vor ...

Markus steht glücklich kurz vor der Fertigstellung seines coolen neuen Java-Programms. Wochenlang war er im »Nur-noch-ein-letztes-Mal-kompilieren«-Modus – aber jetzt ist er tatsächlich fertig. Das Programm ist eine ziemlich raffinierte GUI-Anwendung, aber da es zum größten Teil aus Swing-Code besteht, hat er nur neun Klassen selbst geschrieben.

Endlich ist es so weit, das Programm kann nun an den Kunden geliefert werden. Markus glaubt, er müsse nichts weiter tun, als die neun class-Dateien zu kopieren, da die Java-API beim Kunden bereits installiert ist. Er beginnt damit, sich den Inhalt des Verzeichnisses auflisten zu lassen, in dem all seine Dateien liegen ...

Huch! Es ist was Seltsames passiert. Er sieht nicht 18 Dateien (9 Quellcode-Dateien und 9 kompilierte class-Dateien), sondern 31, von denen viele sehr merkwürdige Namen haben wie:

Konto$DateiListener.class

Diagramm$SpeicherListener.class

... und so weiter. Er hatte völlig vergessen, dass der Compiler auch class-Dateien für all die inneren Klassen der GUI-Event-Listener, die er geschrieben hat, erzeugen muss – genau darum handelt es sich nämlich bei den Klassen mit den komischen Namen.

Jetzt muss er sorgfältig alle class-Dateien herauspicken, die er braucht. Wenn er auch nur eine vergisst, wird sein Programm nicht funktionieren. Aber das ist knifflig, denn es liegt alles unordentlich durcheinander im gleichen Verzeichnis, und er will dem Kunden ja nicht versehentlich eine seiner *Quellcode*-Dateien mitschicken.

# Trennen Sie Quellcode und class-Dateien

Ein einziges Verzeichnis mit Quellcode- und class-Dateien ist Kuddelmuddel. Wie sich herausstellt, hätte Markus seine Dateien von Anfang an durch Trennung von Quellcode und kompiliertem Code organisieren sollen. Mit anderen Worten, er hätte dafür sorgen müssen, dass seine kompilierten class-Dateien nicht im gleichen Verzeichnis landen wie sein Quellcode.

***Die Lösung liegt in der Kombination aus einer bestimmten Verzeichnisstruktur und der Compiler-Option -d.***

Es gibt Dutzende von Möglichkeiten, wie man seine Dateien organisieren kann, und möglicherweise gibt es bei Ihrer Firma eine spezielle Art und Weise, die vorgeschrieben ist. Wir empfehlen Ihnen jedoch ein Organisationsschema, das schon fast zum Standard geworden ist.

Bei diesem Schema erzeugen Sie ein Projektverzeichnis und darin wiederum ein Verzeichnis mit dem Namen **source** und eins mit dem Namen **classes**. Sie beginnen mit dem Abspeichern Ihres Quellcodes (der java-Dateien) im **source**-Verzeichnis. Der Trick besteht dann darin, Ihren Code so zu kompilieren, dass das Ergebnis (die .class-Dateien) im Verzeichnis **classes** landet.

Und für diesen Zweck gibt es einen netten Compiler-Schalter, `-d` (das `d` steht für »directory«, was auf Deutsch »Verzeichnis« bedeutet).

## Mit dem Verzeichnisschalter `-d` kompilieren

```
%cd MeinProjekt/source
%javac -d ../classes MeineAnw.java
```

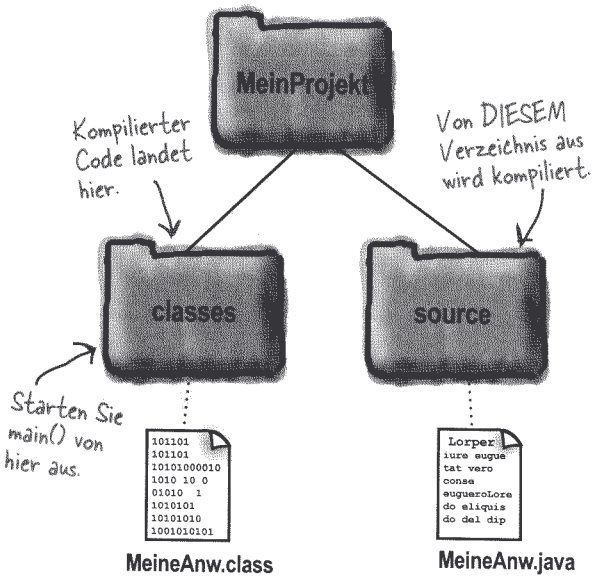
Sagt dem Compiler, er soll den kompilierten Code (die class-Dateien) im Verzeichnis >>classes<< speichern. Dafür muss er vom aktuellen Arbeitsverzeichnis aus ein Verzeichnis nach oben und dann wieder eins nach unten gehen.

Als Letztes steht weiterhin der Name der zu kompilierenden java-Datei.

Mit dem `-d`-Schalter entscheiden *Sie,* in welchem ***Verzeichnis*** der kompilierte Code landet, anstatt die Voreinstellung zu übernehmen, nach der class-Dateien im gleichen Verzeichnis landen wie der Quellcode. Um *alle* java-Dateien im source-Verzeichnis zu kompilieren, verwenden Sie:

```
%javac -d ../classes *.java
```

*.java kompiliert ALLE Quellcode-Dateien im aktuellen Verzeichnis.




## Code ausführen

```
%cd MeinProjekt/classes
%java MeineAnw
```

Starten Sie Ihr Programm aus dem Verzeichnis >>classes<<.

(Anmerkung für den Fall, dass Probleme auftreten: In diesem Kapitel setzen wir immer voraus, dass das aktuelle Arbeitsverzeichnis (d.h. >>.<<) in Ihrem Klassenpfad liegt. Wenn Sie explizit eine classpath-Umgebungsvariable gesetzt haben, vergewissern Sie sich, dass >>.<< darin enthalten ist.)

# Stecken Sie Ihr Java in ein JAR

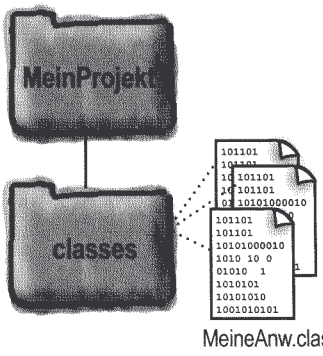
Eine **JAR**-Datei ist ein **J**ava-**AR**chiv. In diesem auf dem pkzip-Dateiformat basierenden »Gefäß« (engl. *jar*) können Sie all Ihre Klassen bündeln und so Ihrem Kunden statt 31 class-Dateien nur eine einzige JAR-Datei aushändigen. Wenn Sie mit dem tar-Befehl unter Unix vertraut sind, werden Sie die Befehle des jar-Werkzeugs wiedererkennen. (Anmerkung: Wenn wir JAR in Großbuchstaben schreiben, beziehen wir uns auf die *Archivdatei.* Schreiben wir es klein, meinen wir das *jar-Werkzeug,* mit dem Sie JAR-Dateien erzeugen.)

Die Frage ist, was *macht* der Kunde mit dem JAR? Wie bringen Sie es dazu, dass es *läuft?*

Sie machen das JAR ***ausführbar.***

Ein ausführbares JAR bedeutet, dass der Endanwender die class-Dateien nicht herausholen muss, bevor er das Programm ausführt. Er kann das Programm ausführen, während die class-Dateien noch im JAR liegen. Der Trick besteht darin, eine ***manifest***-Datei zu erzeugen, die mit in das JAR kommt und Informationen über die Dateien im JAR enthält. Um ein JAR ausführbar zu machen, muss die manifest-Datei der JVM sagen, *welche Klasse die Methode main() enthält!*

## Ein ausführbares JAR machen




**① Sorgen Sie dafür, dass all Ihre class-Dateien im Verzeichnis classes liegen.**

Wir gehen darauf ein paar Seiten später noch genauer ein, aber vorläufig belassen Sie alle class-Dateien in dem Verzeichnis namens »classes«.

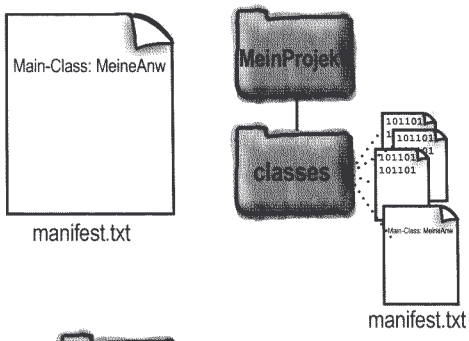
**② Erzeugen Sie eine Datei manifest.txt, die angibt, welche Klasse die main()-Methode enthält.**

Erstellen Sie eine Textdatei namens manifest.txt, die aus nur einer Zeile besteht:

```
Main-Class: MeineAnw
```

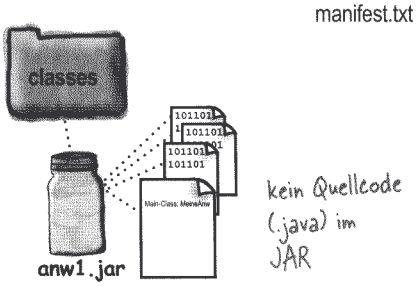
Schreiben Sie nicht .class dahinter.

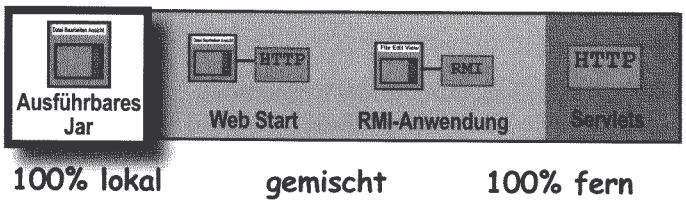
Drücken Sie nach der Eingabe der Main-Class-Zeile die Eingabetaste, sonst funktioniert Ihr »Manifest« möglicherweise nicht korrekt. Legen Sie die manifest-Datei ins Verzeichnis »classes«.




**③ Erzeugen Sie mit dem jar-Werkzeug eine JAR-Datei, die das vollständige class-Verzeichnis enthält, einschließlich der manifest-Datei.**

```
%cd MeinProjekt/classes
%jar -cvmf manifest.txt anw1.jar *.class
ODER
%jar -cvmf manifest.txt anw1.jar MeineAnw.class
```



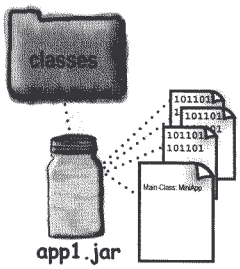



Die meisten 100%ig lokalen Java-Anwendungen kommen als ausführbare JAR-Dateien zum Einsatz.

## Ausführung des JAR

Java (die virtuelle Maschine) ist in der Lage, eine Klasse aus einem JAR zu laden und die main()-Methode dieser Klasse aufzurufen. Tatsächlich kann die gesamte Anwendung im JAR *verbleiben*. Ist die Kugel erst einmal ins Rollen gekommen (d.h. wurde die Methode main() gestartet), ist es der JVM egal, *wo* Ihre Klassen herkommen, solange sie diese überhaupt findet. Und zu den Orten, an denen die JVM nachsieht, gehören auch alle JAR-Dateien im Klassenpfad. Wenn die JVM ein JAR *sehen* kann, *schaut* sie auch darin *nach*, sobald sie eine Datei suchen und laden muss.

Die JVM muss das JAR »sehen«, daher muss es in Ihrem Klassenpfad liegen. Am einfachsten können Sie Ihr JAR sichtbar machen, wenn Sie es direkt im Arbeitsverzeichnis unterbringen.




```
%cd MeinProjekt/classes

%java -jar anw1.jar
```

Der Schalter -jar sagt der JVM, dass sie ein JAR statt einer Klasse erhält.

Die JVM sucht in diesem JAR nach einer manifest-Datei mit einem Einstiegspunkt für Main-Class. Wenn sie nichts findet, gibt es eine Laufzeit-Exception.

Abhängig von der Konfiguration Ihres Betriebssystems ist es eventuell sogar möglich, das JAR einfach mit einem Doppelklick zu starten. Dies funktioniert bei den meisten Windows-Systemen und Mac OS X. Normalerweise können Sie das herbeiführen, indem Sie das JAR auswählen und dem Betriebssystem sagen: »Öffnen mit ...« (oder wie immer die Entsprechung auf Ihrem Betriebssystem heißt).

### Es gibt keine Dummen Fragen

**F: Warum kann ich nicht einfach ein gesamtes Verzeichnis in ein JAR stecken?**

A: Die JVM wirft einen Blick in das JAR und erwartet, dort direkt das zu finden, was sie braucht. Sie wühlt sich nicht erst durch andere Verzeichnisse – es sei denn, die Klasse ist Teil eines Pakets. Und selbst *dann* sieht die JVM nur in den Verzeichnissen nach, die der Paketangabe entsprechen.

**F: Was haben Sie gerade gesagt?**

A: Sie können Ihre class-Dateien nicht einfach in irgendein willkürliches Verzeichnis legen und sie auf diese Weise in ein JAR packen. Aber wenn Ihre Klassen zu Paketen gehören, können Sie die gesamte Paketverzeichnisstruktur in ein JAR stecken. Das *müssen* Sie sogar. Wir erklären das alles auf der nächsten Seite, also können Sie sich erst mal entspannen.

# Stecken Sie Ihre Klassen in Pakete!

Nun haben Sie also ein paar wunderbar wiederverwendbare class-Dateien geschrieben und sie in Ihre interne Entwicklungsbibliothek gestellt, damit andere Programmierer sie benutzen können. Jetzt sonnen Sie sich in dem Bewusstsein, einige der (nach Ihrer bescheidenen Meinung) hervorragendsten Beispiele für OO-Entwicklung geliefert zu haben, die je erdacht wurden. Da klingelt das Telefon. Der Anrufer ist außer sich. Zwei von Ihren Klassen haben den gleichen Namen wie die Klassen, die Fred gerade an die Bibliothek geliefert hat. Und da draußen ist die Hölle los, weil die Arbeit der Entwickler durch Namenskonflikte und -unklarheiten lahm gelegt wird.

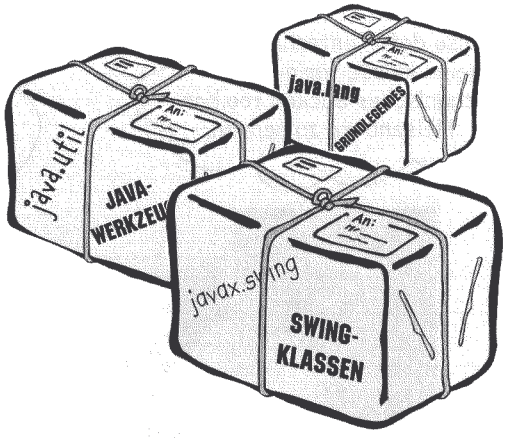
Und das alles, weil *Sie* keine Pakete verwendet haben! Na ja, Sie haben schon Pakete verwendet – in dem Sinn, dass Sie Klassen aus der Java-API benutzt haben, die natürlich in Packages organisiert sind. Aber Ihre eigenen Klassen haben Sie nicht in Pakete gesteckt – und das ist in der echten Welt echt übel.

Wir werden die Organisationsstruktur von den letzten Seiten ein klein wenig modifizieren: Die Klassen stecken wir in ein Paket, und das ganze Paket kommt in das JAR. Passen Sie auch bei den feinen Einzelheiten sehr genau auf; selbst die winzigste Abweichung kann dazu führen, dass Ihr Code nicht mehr kompilierbar und/oder ausführbar ist.

## Pakete verhindern Namenskonflikte

Pakete dienen zwar nicht *nur* dazu, Konflikte zwischen Klassennamen zu verhindern – aber sie dienen *auch* dazu. Beispielsweise könnten Sie eine Klasse namens Kunde, eine namens Konto und eine dritte namens Einkaufswagen schreiben. Und – was sagt man dazu! – wahrscheinlich haben schon die Hälfte aller Entwickler, die im Bereich E-Commerce arbeiten, ebenfalls Klassen mit diesen Namen geschrieben. In einer OO-Welt ist das einfach gefährlich. Wenn eines der Hauptziele von OO im Schreiben wiederverwendbarer Komponenten liegt, müssen Entwickler Komponenten unterschiedlichster Herkunft zusammenstecken und daraus etwas Neues bauen können. Ihre Komponenten müssen »sich gut mit anderen Komponenten vertragen«, auch mit solchen, die Sie nicht geschrieben haben, ja nicht einmal kennen.

Denken Sie daran, wie wir in Kapitel 6 darüber gesprochen haben, dass ein Paketname so etwas ist wie der vollständige Name einer Klasse (der im Englischen als »fully-qualified name« bezeichnet wird). Die Klasse ArrayList heißt eigentlich ***java.util.ArrayList***, JButton eigentlich ***javax.swing.JButton*** und Socket eigentlich ***java.net.Socket***. Beachten Sie, dass zwei dieser Klassen – ArrayList und Socket – mit *java* beginnen. Mit anderen Worten, der erste Teil ihres vollständigen Namens lautet »java«. Denken Sie an eine Hierarchie, wenn Sie an Paketstrukturen denken, und organisieren Sie Ihre Klassen entsprechend.




**Paketstruktur der Java-API für:**

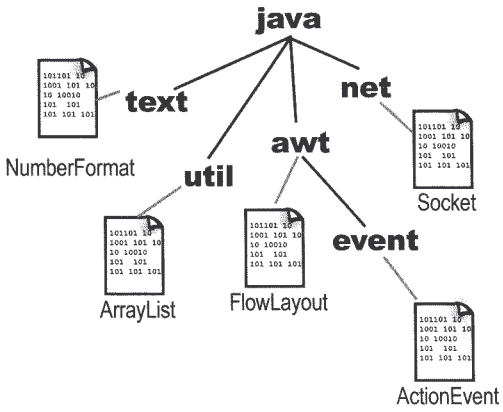
java.text.NumberFormat

java.util.ArrayList

java.awt.FlowLayout

java.awt.event.ActionEvent

java.net.Socket




**Wonach sieht dieses Paket für Sie aus? Sieht es nicht einer Verzeichnishierarchie ziemlich ähnlich?**

## Verhindern von Paketnamenskonflikten

Wenn Sie Ihre Klasse in ein Paket stecken, verringern Sie damit die Gefahr von Namenskonflikten mit anderen Klassen; aber was soll zwei Programmierer davon abhalten, sich identische *Paketnamen* auszudenken? Anders ausgedrückt: Was soll zwei Programmierer, die beide eine Klasse namens Konto haben, daran hindern, die Klasse in ein Paket namens einkauf.kunden zu stecken? In diesem Fall hätten *immer noch* beide Klassen den gleichen Namen:

***einkauf.kunden.Konto***

Sun empfiehlt ausdrücklich eine Paketnamenskonvention, die dieses Risiko stark verringert: der Klasse den umgedrehten eigenen Domainnamen voranzustellen. Wie Sie wissen, sind Domainnamen garantiert einzigartig. Es kann zwei verschiedene Männer geben, die Heinz Erhardt heißen, aber es kann nicht zwei verschiedene Domains mit dem Namen nochngedicht.de geben.

Pakete können Namenskonflikte verhindern, aber nur dann, wenn Sie einen Paketnamen wählen, der garantiert einzigartig ist. Die beste Möglichkeit dafür ist, Ihren umgedrehten Domainnamen an den Anfang des Pakets zu stellen:

de.javavkbf.Buch

Paketname — Klassenname

### Paketnamen mit umgedrehten Domains

de.javavkbf.projekte.Diagramm

Beginnen Sie den Paketnamen mit Ihrer umgedrehten Domain, getrennt durch einen Punkt (.), und fügen Sie dann Ihre Organisationsstruktur an.

projekte.Diagramm könnte ein häufiger Name sein, aber in der Kombination mit de.javavkbf müssen wir uns nur über die anderen Entwickler in der Firma Gedanken machen.

Der Klassenname wird immer mit großem Anfangsbuchstaben geschrieben.

## So stecken Sie Ihre Klasse in ein Paket:

### ① Wählen Sie einen Paketnamen

Wir benutzen als Beispiel **de.javavkbf**. Der Klassenname ist PaketUebung, so dass der vollständige Name der Klasse **de.javavkbf.PaketUebung** lautet.

### ② Schreiben Sie eine Paketangabe in Ihre Klasse

Sie muss als erste Anweisung in der Quellcode-Datei stehen, noch oberhalb der import-Anweisungen. Es darf nur eine Paketangabe pro Quellcode-Datei geben, daher **müssen alle Klassen in einer Quellcode-Datei zum gleichen Paket gehören**. Das gilt natürlich auch für innere Klassen.

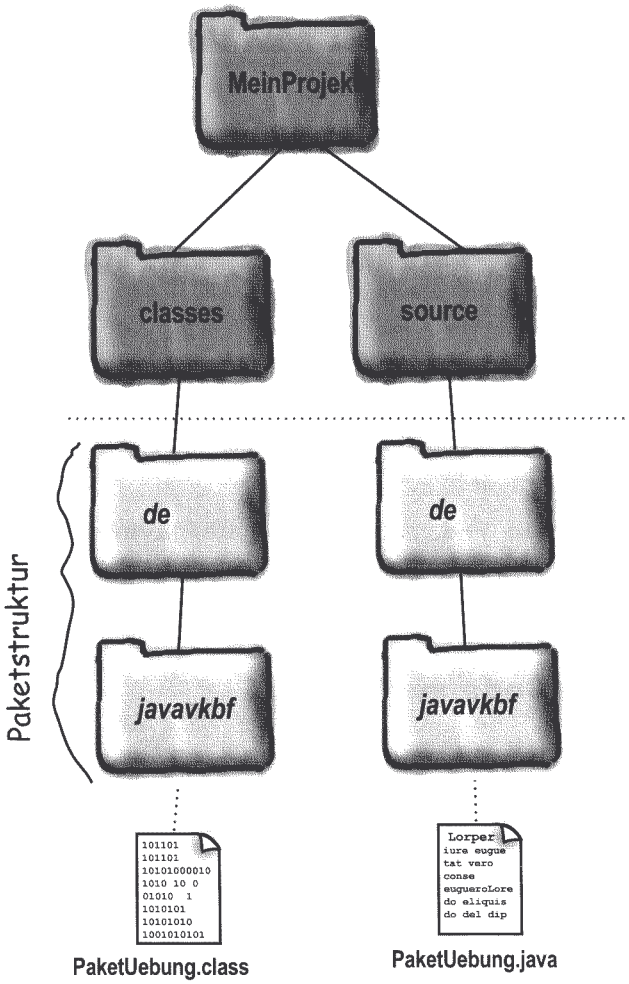
```
package de.javavkbf;

import javax.swing.*;

public class PaketUebung {
  // Code, der Ihr
  // Leben verändert
}
```

### ③ Erstellen Sie eine entsprechende Verzeichnisstruktur

Es genügt nicht, die Paketzugehörigkeit Ihrer Klasse nur zu ***verkünden***, indem Sie einfach eine Paketangabe in den Code schreiben. Ihre Klasse gehört erst dann *wirklich* zu einem Paket, wenn Sie sie in eine passende Verzeichnisstruktur setzen. Lautet also der vollständige Klassenname de.javavkbf.PaketUebung, ***müssen*** Sie den PaketUebung-Quellcode in ein Verzeichnis namens **javavkbf** legen, das wiederum in einem Verzeichnis namens **de** liegen ***muss***.

Wenn Sie das nicht tun, lässt es sich zwar trotzdem kompilieren, aber glauben Sie uns – das ist die anderen Probleme, die Sie sich einhandeln, nicht wert. Bewahren Sie Ihren Quellcode in einer Verzeichnisstruktur auf, die der Paketstruktur entspricht, und vermeiden Sie so Unmengen schlimmer Kopfschmerzen zu einem späteren Zeitpunkt.

**Sie müssen eine Klasse in eine Verzeichnisstruktur legen, die der Pakethierarchie entspricht.**




Erstellen Sie eine übereinstimmende Verzeichnisstruktur für den Quellcode- und den Klassenbaum.

# Kompilieren und Ausführen mit Paketen

Wenn Ihre Klasse in einem Paket liegt, ist das Kompilieren und Ausführen ein wenig komplizierter. Der entscheidende Punkt ist, dass der Compiler und die JVM in der Lage sein müssen, Ihre Klasse und all die anderen von ihr verwendeten Klassen zu finden. Bei den Klassen der Kern-API ist das kein Problem. Wo es seine eigenen Sachen findet, weiß Java immer. Aber das Verfahren, Ihre eigenen Klassen von dem Verzeichnis aus zu kompilieren, in dem die Quellcode-Dateien liegen, wird einfach nicht funktionieren (oder zumindest nicht *zuverlässig*). Wenn Sie so vorgehen, wie wir es auf dieser Seite beschreiben, garantieren wir Ihnen jedoch, dass Sie Erfolg haben werden. Es gibt noch weitere Möglichkeiten, aber für uns hat sich dieser Weg als der zuverlässigste und gangbarste erwiesen.

## Kompilieren mit dem Verzeichnisschalter `-d`

```
%cd MeinProjekt/source
```

Bleiben Sie im source-Verzeichnis! Steigen Sie NICHT mit cd bis in das Verzeichnis hinunter, in dem die .java-Datei liegt!

```
%javac -d ../classes de/javavkbf/PaketUebung.java
```

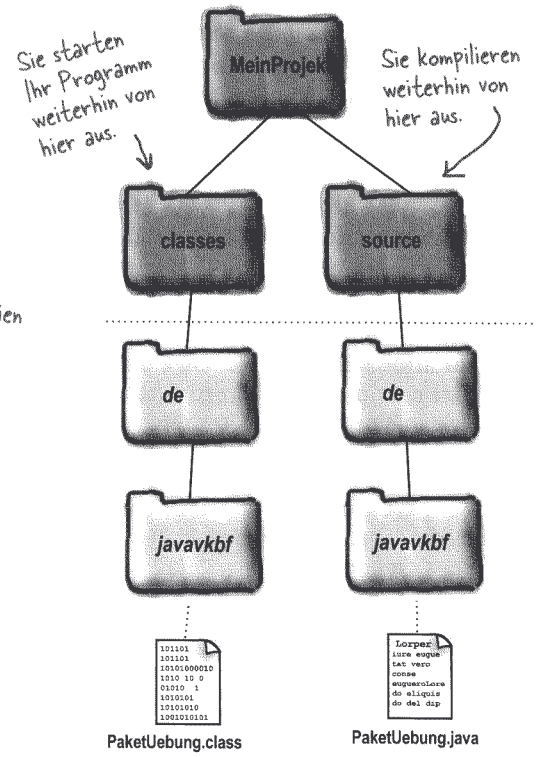
Weist den Compiler an, den kompilierten Code (die class-Dateien) in das Verzeichnis classes zu legen, innerhalb der korrekten Paketstruktur!!! Ja, die kennt er.

Jetzt müssen Sie den PFAD angeben, der zur eigentlichen Quellcode-Datei führt.

Um alle .java-Dateien im Paket de/javavkbf zu kompilieren, schreiben Sie:

```
%javac -d ../classes de/javavkbf/*.java
```

Kompiliert alle Quellcode-Dateien (.java) in diesem Verzeichnis.




## Den Code ausführen

```
%cd MeinProjekt/classes
```

Starten Sie Ihr Programm vom classes-Verzeichnis aus.

```
%java de.javavkbf.PaketUebung
```

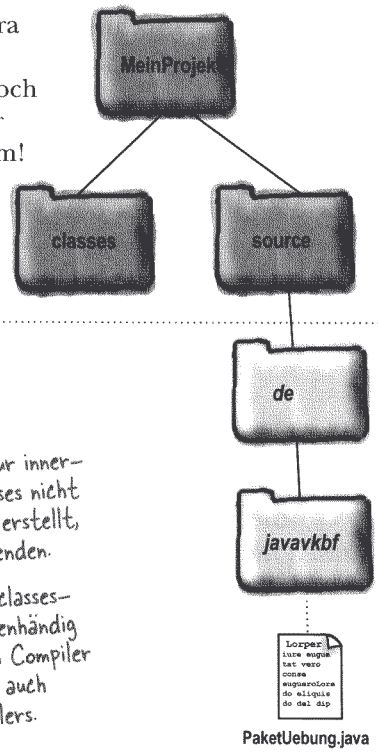
Sie MÜSSEN den vollständigen Klassennamen angeben! Die JVM wird ihn sehen und sofort im aktuellen Verzeichnis (classes) nachschauen, in der Erwartung, dort ein Verzeichnis namens >>de<< zu finden und in diesem wiederum ein Verzeichnis namens >>javavkbf<< und darin die Klasse. Wenn die Klasse im Verzeichnis >>de<< liegt oder gar in >>classes<<, funktioniert es nicht!

# Der Schalter -d kann sogar noch mehr

Das Kompilieren mit dem Schalter `-d` geht wunderbar, weil man damit nicht nur die kompilierte Klasse in ein anderes Verzeichnis schicken kann als das, in dem die Quelldatei liegt, sondern weil `-d` auch weiß, wie es die class-Datei korrekt in die Verzeichnisstruktur für das Paket steckt, zu dem die Klasse gehört.

Aber es kommt noch besser!

Nehmen wir mal an, Sie haben extra für Ihren Quellcode eine hübsche Verzeichnisstruktur erstellt, aber noch keine passende Verzeichnisstruktur für Ihre class-Dateien. Kein Problem! Das Kompilieren mit `-d` weist Ihren Compiler nicht nur an, die Klassen im richtigen Verzeichnisbaum *abzulegen*, sondern auch, die Verzeichnisse zu *erzeugen*, falls sie nicht existieren.




Falls die Paketverzeichnisstruktur innerhalb des »classes«-Verzeichnisses nicht existiert, wird sie vom Compiler erstellt, wenn Sie den -d-Schalter verwenden.

Sie müssen die Verzeichnisse im classes-Wurzelverzeichnis also nicht eigenhändig erzeugen. Und wenn Sie das vom Compiler erledigen lassen, besteht zudem auch nicht die Gefahr eines Tippfehlers.

**Der -d-Schalter sagt dem Compiler: »Leg die Klasse in ihre Paketverzeichnisstruktur mit der hinter dem -d angegebenen Klasse als Wurzelverzeichnis. Aber ... wenn es die Verzeichnisse nicht gibt, erzeug diese erst einmal und leg *dann* die Klasse an den richtigen Platz!«**

## Es gibt keine Dummen Fragen

**F: Ich habe versucht, mit cd in das Verzeichnis zu wechseln, in dem meine main-Klasse liegt, doch jetzt behauptet die JVM, sie könne meine Klasse nicht finden! Dabei liegt sie genau DORT, im aktuellen Verzeichnis!**

A: Wenn Ihre Klasse erst einmal zu einem Paket gehört, können Sie sie nicht mehr mit Ihrem »Kurz«namen aufrufen. Sie MÜSSEN auf der Kommandozeile den vollständigen Namen der Klasse angeben, deren main()-Methode Sie ausführen wollen. Aber da der vollständige Name auch die *Paket*struktur umfasst, besteht Java darauf, dass die Klasse in einer entsprechenden *Verzeichnis*struktur liegt. Wenn Sie also auf der Kommandozeile sagen:

```
%java com.foo.Buch
```

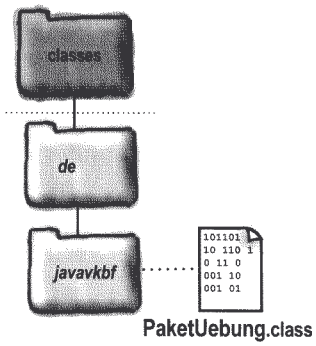
sucht die JVM in ihrem aktuellen Verzeichnis (und im restlichen Klassenpfad) nach einem Verzeichnis namens »com«: ***Sie sucht nicht nach einer Klasse namens »Buch«, bevor sie nicht ein Verzeichnis namens »com« und darin ein Verzeichnis namens »foo« gefunden hat.*** Erst dann akzeptiert die JVM, dass sie die korrekte Klasse Buch gefunden hat. Wenn sie an irgendeiner anderen Stelle eine Klasse Buch findet, nimmt sie an, dass sich diese Klasse nicht in der richtigen Struktur befindet – selbst wenn das doch der Fall sein sollte. Die JVM wird beispielsweise nicht entlang des Verzeichnisbaums zurückschauen und sagen: »Oh, ich sehe, über uns ist ein Verzeichnis namens com, also muss das hier das richtige Paket sein ...«

# Ein ausführbares JAR mit Paketen erzeugen

Wenn Ihre Klasse zu einem Paket gehört, muss es diese Paketverzeichnisstruktur auch innerhalb des JAR geben! Sie können Ihre Klassen nicht einfach so in das JAR stopfen, wie wir es in unseren Vorpaketzeiten getan haben. Und es dürfen keine anderen Verzeichnisse oberhalb Ihres Pakets mit hineingelangen. Das erste Verzeichnis im Paket (normalerweise de, at, ch, com o.Ä.) muss das erste Verzeichnis innerhalb des JAR sein! Wenn Sie versehentlich das Verzeichnis *oberhalb* des Pakets mit einschließen (z.B. »classes«), wird das JAR nicht richtig funktionieren.

## Ein ausführbares JAR erzeugen

**① Vergewissern Sie sich, dass all Ihre class-Dateien innerhalb der richtigen Paketstruktur liegen, also unterhalb des classes-Verzeichnisses.**

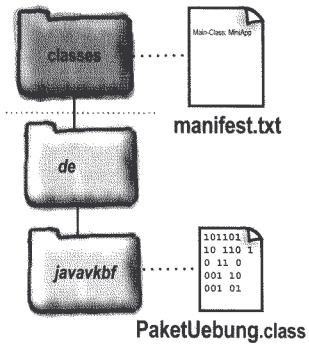



**② Erzeugen Sie die Datei manifest.txt, die angibt, welche Klasse die main()-Methode enthält, und verwenden Sie unbedingt den vollständigen Klassennamen!**

Erzeugen Sie eine Textdatei namens manifest.txt mit einer einzigen Zeile:

```
Main-Class: de.javavkbf.PaketUebung
```

Legen Sie die manifest-Datei in das Verzeichnis »classes«.

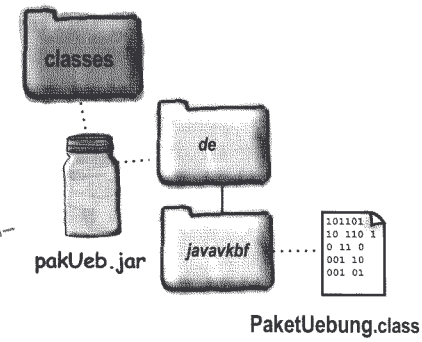



**③ Führen Sie das jar-Werkzeug aus und erzeugen Sie damit eine JAR-Datei, die die Paketverzeichnisse und das »Manifest« enthält.**

Sie brauchen nichts weiter als das Verzeichnis »de« anzugeben, und schon wird das gesamte Paket (mit allen Klassen) in das JAR gesteckt.

```
%cd MeinProjekt/classes

jar -cvmf manifest.txt pakUeb.jar de
```

Sie geben lediglich das Verzeichnis de an und bekommen alles mit hineingepackt!



# Was ist aus der manifest-Datei geworden?

Warum werfen wir nicht mal einen Blick in das JAR und finden es heraus? Man kann auf der Kommandozeile mit dem jar-Werkzeug nicht nur ein JAR erzeugen und ausführen, man kann auch den Inhalt eines JAR extrahieren (so wie man ein zip- oder tar-Archiv entpackt).

Stellen Sie sich vor, Sie haben das pakUeb.jar in ein Verzeichnis namens Julia gelegt.

## jar-Kommandos zum Auflisten und Extrahieren

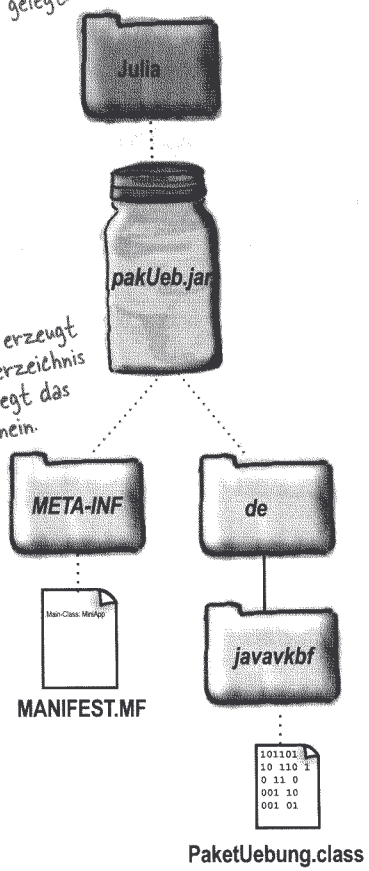
### ① Den Inhalt eines JAR auflisten

```
% jar -tf pakUeb.jar
```

-tf steht für »Table File«; der Inhalt der JAR-Datei soll tabellarisch aufgelistet werden.

```
Datei Bearbeiten Fenster Hilfe EssiggurkenImGlas
% cd Julia
% jar -tf pakUeb.jar
META-INF/
META-INF/MANIFEST.MF
de/
de/javavkbf/
de/javavkbf/PaketUebung.class
```

Wir haben die JAR-Datei in ein Verzeichnis namens Julia gelegt.

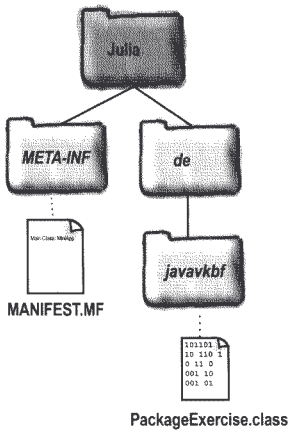



Das jar-Werkzeug erzeugt automatisch ein Verzeichnis META-INF und legt das Manifest dort hinein.

### ② Den Inhalt eines JAR extrahieren

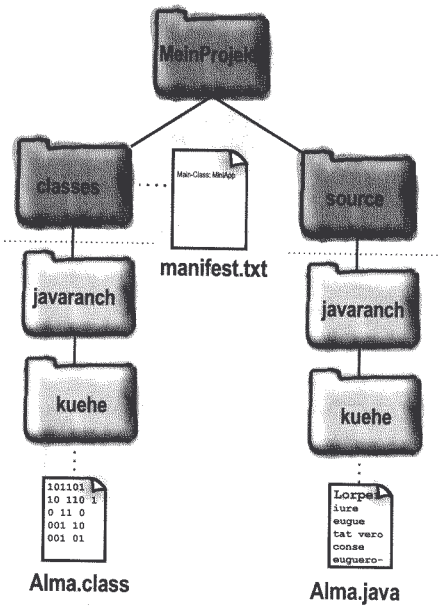
```
% cd Julia
%jar -xf pakUeb.jar
```

-xf steht für »Extract File« und funktioniert genau wie das Entpacken eines zip- oder tar-Archivs. Wenn Sie die Datei pakUeb.jar extrahieren, finden Sie die Verzeichnisse META-INF und de in Ihrem aktuellen Verzeichnis.




META-INF steht für »Meta-information«. Das jar-Werkzeug erzeugt sowohl das Verzeichnis META-INF als auch die Datei MANIFEST.MF. Außerdem nimmt es den Inhalt *Ihrer* manifest-Datei und schreibt ihn in die Datei MANIFEST.MF. *Ihre* manifest-*Datei* kommt also nicht mit in das JAR, aber der *Inhalt* wird in das »echte« Manifest (MANIFEST.MF) geschrieben.

## Spitzen Sie Ihren Bleistift




Finden Sie heraus, was Sie für die Paket/Verzeichnis-Struktur in diesem Bild auf der Kommandozeile eingeben müssen, um den Code zu kompilieren, auszuführen, ein JAR zu erzeugen und ein JAR auszuführen. Nehmen Sie an, wir halten uns an den Standard, bei dem die Verzeichnisstruktur direkt unterhalb von *source* und *classes* beginnt. Anders ausgedrückt: Die Verzeichnisse *source* und *classes* sind nicht Teil des Pakets.

### Kompilieren:

```
%cd source
%javac _____
```

### Ausführen:

```
%cd _____

%java _____
```

### Ein JAR erzeugen

```
%cd _____

% _____
```

### Ein JAR ausführen

```
%cd _____

% _____
```

Zusatzfrage: Was ist an dem Paketnamen falsch?

## Es gibt keine Dummen Fragen

**F: Was passiert, wenn Sie ein ausführbares JAR laufen lassen wollen und beim Endanwender kein Java installiert ist?**

A: Es wird nicht funktionieren, weil Java-Code ohne JVM nicht ausgeführt werden kann. Beim Endanwender muss Java installiert sein.

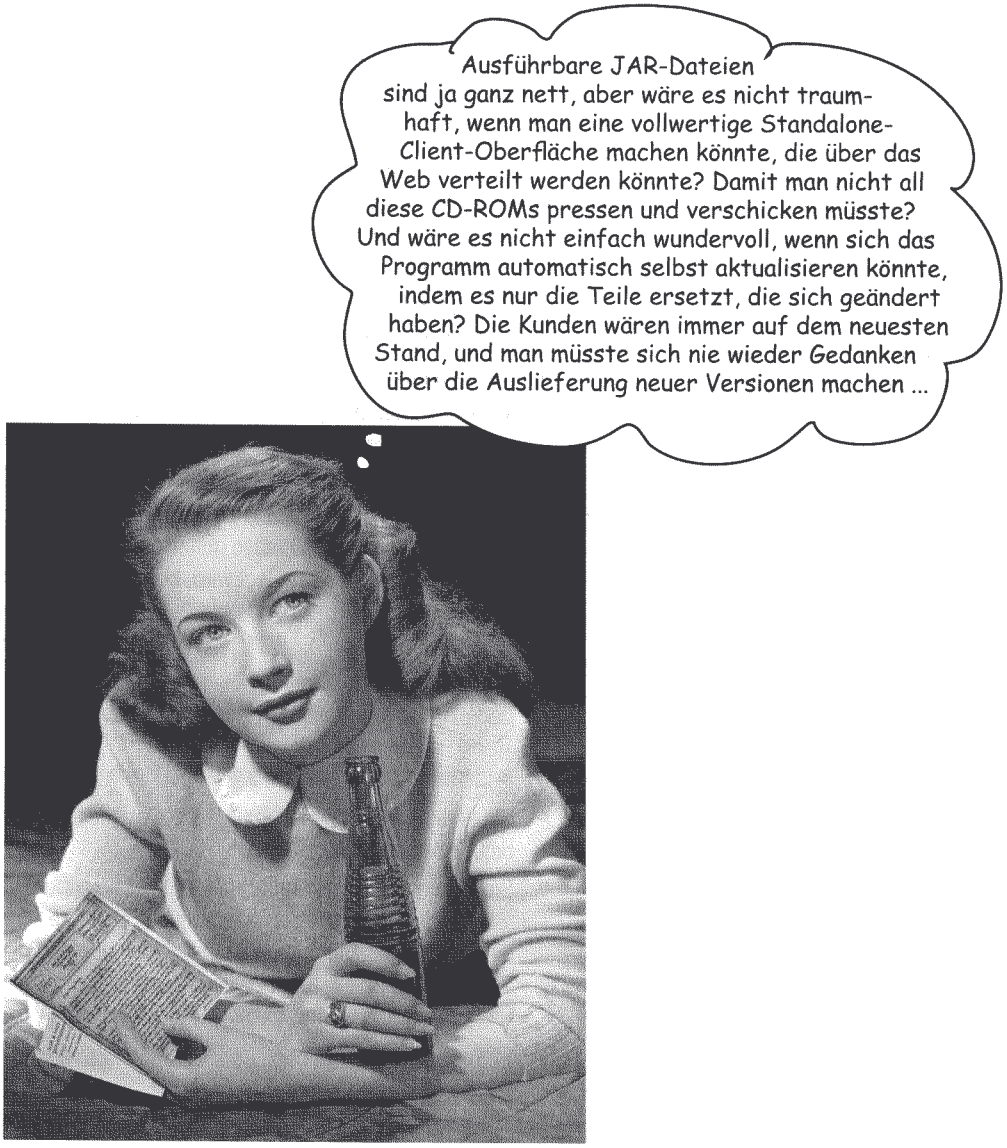
**F: Wie kann ich dafür sorgen, dass Java auf dem Rechner des Endanwenders installiert ist?**

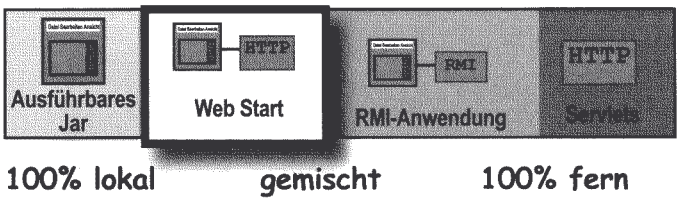
A: Idealerweise können Sie einen eigenen Installer erzeugen und ihn gemeinsam mit Ihrer Anwendung ausliefern. Verschiedene Firmen bieten Installationsprogramme an, die von »einfach« bis »enorm leistungsfähig« reichen. Ein Installationsprogramm könnte beispielsweise herausfinden, ob beim Endanwender eine geeignete Java-Version installiert ist, und falls dies nicht der Fall ist, kann es vor der Installation Ihrer Anwendung Java installieren und konfigurieren. Java-Installationslösungen werden u.a. von Installshield, InstallAnywhere und DeployDirector angeboten.

Was besonders toll ist: Mit einigen der Installationsprogramme können Sie sogar eine Deployment-CD-ROM erstellen, die Installer für alle wichtigen Java-Plattformen enthält. Wenn beim Anwender beispielsweise Solaris läuft, wird die Solaris-Version von Java installiert, unter Windows die Windows-Version usw. Reicht Ihr Budget dafür aus, ist dies bei Weitem der einfachste Weg für Ihre Endanwender, die richtige Java-Version zu installieren und zu konfigurieren.

## Punkt für Punkt

- Organisieren Sie Ihr Projekt so, dass Ihr Quellcode und die class-Dateien nicht im gleichen Verzeichnis liegen.
- Eine Standard-Organisationsstruktur ist es, ein *Projekt*verzeichnis zu erzeugen und in dieses ein Verzeichnis *source* und ein Verzeichnis *classes* zu legen.
- Bei der Organisation Ihrer Klassen in Paketen verhindern Sie Namenskonflikte mit anderen Klassen, wenn Sie einem Klassennamen den umgedrehten Domainnamen voranstellen.
- Um eine Klasse in ein Paket einzubinden, setzen Sie eine package-Anweisung oben in die Quellcode-Datei, noch vor die Importanweisungen:
  `package com.wickedlysmart;`
- Um zu einem Paket zu gehören, muss eine Klasse *in einer Verzeichnisstruktur liegen, die genau der Paketstruktur entspricht.* Bei einer Klasse com.wickedlysmart.Foo muss die Klasse Foo in einem Verzeichnis namens *wickedlysmart* liegen, das wiederum in einem Verzeichnis namens *com* liegt.
- Damit Ihre kompilierte Klasse in der richtigen Paketverzeichnisstruktur im Verzeichnis *classes* abgelegt wird, benutzen Sie den Compiler-Schalter `-d`:
  ```
  % cd source
  % javac -d ../classes com/wickedlysmart/Foo.java
  ```
- Um Ihren Code auszuführen, wechseln Sie in das Verzeichnis classes und geben den vollständigen Namen Ihrer Klasse an:
  ```
  % cd classes
  % java com.wickedlysmart.Foo
  ```
- Sie können Ihre Klassen in JAR-Dateien (Java-ARchiven) bündeln. JAR basiert auf dem pkzip-Format.
- Sie können eine ausführbare JAR-Datei erzeugen, indem Sie ein »Manifest« in das JAR einfügen, das angibt, in welcher Klasse die main()-Methode zu finden ist. Um eine manifest-Datei zu erzeugen, erstellen Sie eine Textdatei mit einem Eintrag wie diesem:
  `Main-Class: com.wickedlysmart.Foo`
- Vergessen Sie nicht, nach dem Eintippen der Main-Class-Zeile die Eingabetaste zu drücken, sonst funktioniert Ihre manifest-Datei möglicherweise nicht.
- Um eine JAR-Datei zu erzeugen, geben Sie ein:
  `jar -cvmf manifest.txt MeinJar.jar com`
- Die gesamte Paketverzeichnisstruktur (und *nur* die Verzeichnisse, die dem Paket entsprechen) muss unmittelbar innerhalb der JAR-Datei liegen.
- Um eine ausführbare JAR-Datei zu starten, geben Sie ein:
  `java -jar MeinJar.jar`


Ausführbare JAR-Dateien sind ja ganz nett, aber wäre es nicht traumhaft, wenn man eine vollwertige Standalone-Client-Oberfläche machen könnte, die über das Web verteilt werden könnte? Damit man nicht all diese CD-ROMs pressen und verschicken müsste? Und wäre es nicht einfach wundervoll, wenn sich das Programm automatisch selbst aktualisieren könnte, indem es nur die Teile ersetzt, die sich geändert haben? Die Kunden wären immer auf dem neuesten Stand, und man müsste sich nie wieder Gedanken über die Auslieferung neuer Versionen machen ...




# Java Web Start

Mit Java Web Start (JWS) wird Ihre Anwendung beim ersten Mal aus einem Webbrowser heraus gestartet (daher der Name Java *Web Start)*; sie läuft jedoch als Stand-alone-Anwendung (na ja, zumindest *fast*) ohne Einschränkungen durch den Browser. Und ist sie erst einmal auf den Rechner des Endanwenders heruntergeladen (was gleich beim erstmaligen Klick des Anwenders, der den Download startet, auf den Link im Browser passiert), *bleibt* sie auch dort.

Java Web Start ist – unter anderem – ein kleines Java-Programm, das auf dem Client-Rechner wohnt und ganz ähnlich wie ein Browser-Plug-in funktioniert (so wie z.B. der Acrobat Reader, der sich öffnet, wenn Ihr Browser eine pdf-Datei erhält). Man bezeichnet dieses Java-Programm auch als **Java Web Start-Hilfsanwendung**; ihre wichtigste Aufgabe ist es, das Herunterladen, die Aktualisierung und den Start (die Ausführung) *Ihrer* JWS-Anwendungen zu steuern.

Wenn JWS Ihre Anwendung (ein ausführbares JAR) herunterlädt, ruft es deren main()-Methode auf. Danach kann der Endanwender Ihre Anwendung direkt von der JWS-Hilfsanwendung aus starten, *ohne* erneut über den Link auf der Webseite gehen zu müssen.

Aber es kommt noch besser. Das Erstaunliche an JWS ist seine Fähigkeit, es zu merken, wenn sich auch nur ein kleiner Teil des Programms (z.B. eine einzelne Klassendatei) auf dem Server geändert hat, und den aktualisierten Code dann herunterzuladen und einzubauen, ohne dass der Endanwender eingreifen muss.

Da gibt es natürlich trotzdem noch ein Problem: Wie bekommen die Endanwender Java und Java Web Start? Sie brauchen beides – Java für die Ausführung des Programms und Java Web Start (selbst ein kleines Java-Programm), um damit das Programm abzurufen und zu starten. Doch selbst *dafür* gibt es eine Lösung. Sie können es so einrichten, dass Ihr Endanwender sich JWS, falls er es nicht hat, von Sun herunterladen kann. Und falls er es doch hat, aber seine Java-Version veraltet ist (weil Sie in Ihrer JWS-Anwendung angegeben haben, dass Sie eine bestimmte Java-Version benötigen), kann auch die Java 2 Standard Edition auf den Rechner des Endanwenders heruntergeladen werden.

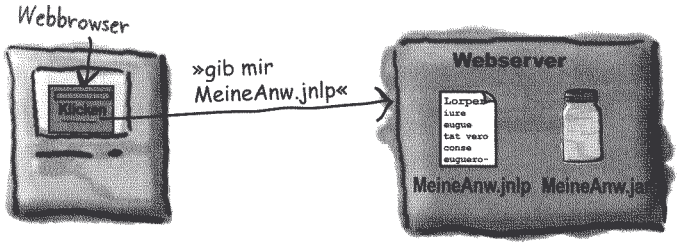
Und das Beste ist: Es ist einfach zu benutzen. Sie können eine JWS-Anwendung ganz wie jede andere Art von Webressource anbieten, wie z.B. eine gute alte HTML-Seite oder ein JPEG-Bild. Sie erstellen eine Webseite (HTML-Seite) mit einem Link auf Ihre JWS-Anwendung, und schon sind Sie im Geschäft.

Letztendlich ist Ihre JWS-Anwendung nichts weiter als ein ausführbares JAR, das sich Endanwender aus dem Web herunterladen können.

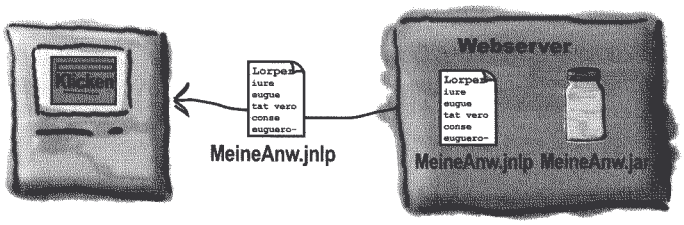
Endanwender starten eine Java Web Start-Anwendung durch das Anklicken eines Links auf einer Webseite. Ist die Anwendung dann einmal heruntergeladen, läuft sie außerhalb des Browsers, genau wie jede andere Stand-alone-Java-Anwendung. Tatsächlich ist eine Java Web Start-Anwendung nichts weiter als ein ausführbares JAR, das über das Web verteilt wird.

## Wie Java Web Start funktioniert

1. **Der Client klickt auf einen Webseitenlink zu Ihrer JWS-Anwendung (einer jnlp-Datei).**
   Der Webseitenlink:
   ```
   <a href="MeineAnw.jnlp">Klicken</a>
   ```

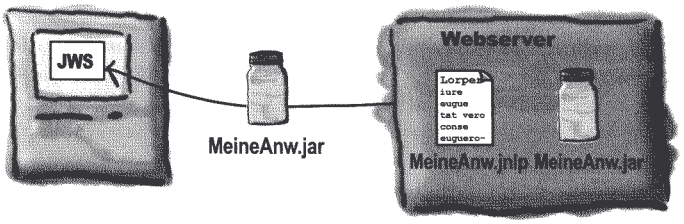



2. **Der Webserver (HTTP) erhält die Anfrage und schickt eine jnlp-Datei herüber (dies ist NICHT das JAR).**
   Die jnlp-Datei ist ein XML-Dokument, in dem der Name der ausführbaren JAR-Datei für dieses Programm angegeben ist.

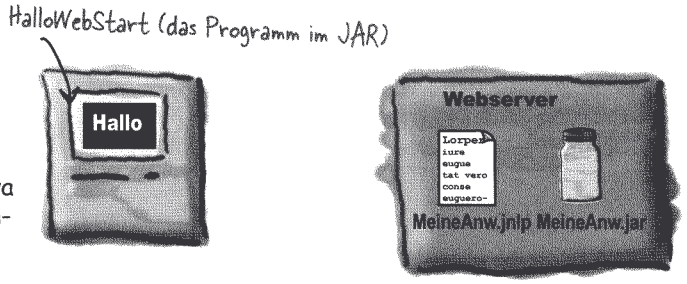



3. **Java Web Start (eine kleine Hilfsanwendung auf dem Client) wird vom Browser gestartet. Die JWS-Hilfsanwendung liest die jnlp-Datei und bittet den Server um die Datei MeineAnw.jar.**

4. **Der Webserver »serviert« die angeforderte .jar-Datei.**




5. **Java Web Start erhält das JAR und startet das Programm durch Aufruf der angegebenen main()-Methode (genau wie ein ausführbares JAR).**
   Wenn der Anwender das nächste Mal dieses Programm ausführen möchte, kann er das Java Web Start-Programm öffnen und Ihre Anwendung von dort aus starten, ohne online sein zu müssen.



# Die .jnlp-Datei

Für eine Java Web Start-Anwendung brauchen Sie eine .jnlp-Datei (jnlp steht für »Java Network Launch Protocol«), die Ihr Programm beschreibt. Die JWS-Anwendung liest diese Datei, findet damit Ihr JAR und startet Ihr Programm (durch Aufruf der main()-Methode im JAR). Eine jnlp-Datei ist ein einfaches XML-Dokument, in dem ganz unterschiedliche Dinge stehen können; zumindest sollte es jedoch Folgendes enthalten:

```
<?xml version="1.0" encoding="utf-8"?>

<jnlp spec="0.2 1.0"
    codebase="http://127.0.0.1/~kathy"
    href="MeineAnw.jnlp">

  <information>
    <title>Kathys Anwendung</title>
    <vendor>Wickedly Smart</vendor>
    <homepage href="index.html"/>
    <description>Von Kopf bis Fuss WebStart Demo</description>
    <icon href="kathys.gif"/>
    <offline-allowed/>
  </information>

  <resources>
    <j2se version="1.3+"/>
    <jar href="MeineAnw.jar"/>
  </resources>

  <application-desc main-class="HalloWebStart"/>
</jnlp>
```

Im »codebase«-Tag geben Sie an, unter welchem »Wurzel«verzeichnis auf dem Server Ihre Web Start-Sachen liegen. Wir testen dies auf unserem Localhost, daher setzen wir hier die lokale Prüfschleifenadresse (engl. Loopback-Adresse) »127.0.0.1« ein. Bei Web Start-Anwendungen auf unserem Internet-Webserver würde es hier heißen: »http://www.wickedlysmart.com«.

Dies gibt an, wo die jnlp-Datei bezogen auf die »Codebasis« liegt. In unserem Beispiel ist MeineAnw.jnlp direkt im angegebenen Basisverzeichnis des Webservers zu finden, nicht in irgendeinem untergeordneten Verzeichnis.

Diese Tags müssen alle vorhanden sein, sonst funktioniert Ihr Programm möglicherweise nicht richtig! Die »information«-Tags werden von der JWS-Hilfsanwendung benutzt, vor allem für die Anzeige, wenn der Benutzer eine zuvor heruntergeladene Anwendung neu starten möchte.

Dies bedeutet, dass der Benutzer Ihr Programm auch ohne Verbindung mit dem Internet starten kann. Wenn er offline ist, wird allerdings das automatische Update nicht funktionieren.

Dies heißt, dass unser Programm Java in der Version 1.3 oder höher benötigt.

Der Name Ihres ausführbaren JAR! Möglicherweise haben Sie zusätzlich noch andere JAR-Dateien mit weiteren Klassen oder gar mit Sounds und Bildern, die Ihr Programm verwendet.

Dies entspricht dem Manifest-Main-Class-Eintrag ... es gibt an, welche Klasse im JAR die main()-Methode enthält (an dieser Stelle haben wir auf die Angabe von Paketen verzichtet).

# Die Schritte bei der Erstellung und beim Deployment einer Java Web Start-Anwendung

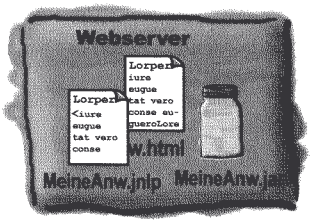
1. Erzeugen Sie ein ausführbares JAR für Ihre Anwendung.

MeineAnw.jar

2. Schreiben Sie eine jnlp-Datei.

MeineAnw.jnlp

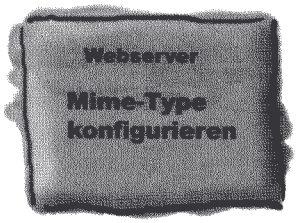
3. Stellen Sie Ihr JAR und die .jnlp-Datei auf Ihren Webserver.




4. Fügen Sie für Ihren Webserver einen neuen Mime-Typ hinzu.

`application/x-java-jnlp-file`

Dadurch schickt Ihr Server die jnlp-Datei mit dem richtigen Header, damit der Browser beim Empfang der jnlp-Datei weiß, was das ist und dass die JWS-Hilfsanwendung gestartet werden muss.



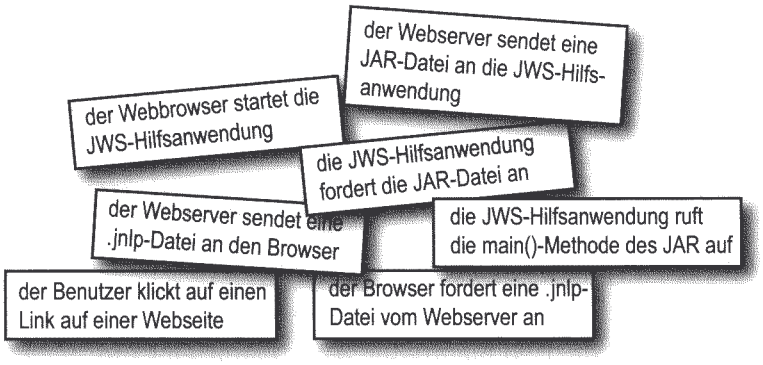

5. Erstellen Sie eine Webseite mit einem Link auf Ihre .jnlp-Datei.

MeineJWSAnw.html

```
<HTML>
  <BODY>
    <a href="MeineAnw.jnlp">Meine Anwendung starten</a>
  </BODY>
</HTML>
```

## Was kam zuerst?

Sehen Sie sich die folgenden Ereignisse an und bringen Sie sie in die Reihenfolge, in der sie in einer JWS-Anwendung eintreten.




1. 
2. 
3. 
4. 
5. 
6. 
7. 

## Es gibt keine Dummen Fragen

**F: Wodurch unterscheidet sich Java Web Start von einem Applet?**

A: Applets können nicht außerhalb eines Webbrowsers existieren. Ein Applet wird aus dem Web nicht einfach von einer Webseite, sondern als Bestandteil einer Webseite geladen. Mit anderen Worten: Für den Browser ist das Applet genau wie ein JPEG oder irgendeine andere Ressource. Zur Ausführung des Applets benutzt der Browser entweder ein Java-Plug-in oder das im Browser eingebaute Java (heutzutage wesentlich seltener). Applets haben bei solchen Dingen wie z.B. automatischer Aktualisierung nicht das gleiche Funktionsniveau, und sie müssen immer aus dem Browser heraus gestartet werden. Wenn JWS-Anwendungen erst einmal aus dem Web heruntergeladen sind, muss der Benutzer gar keinen Browser mehr verwenden, um das Programm lokal neu zu starten. Stattdessen kann er die JWS-Hilfsanwendung aufrufen und damit das bereits heruntergeladene Programm neu starten.

**F: Welche Sicherheitseinschränkungen gibt es bei JWS?**

A: Es gibt bei JWS-Anwendungen verschiedene Einschränkungen; dazu gehört z.B., dass nicht auf der Festplatte des Benutzers gelesen und geschrieben werden darf. Aber JWS besitzt eine eigene API mit einem speziellen Dialog zum Öffnen und Speichern; mit Erlaubnis des Benutzers kann Ihr Programm daher seine eigenen Dateien in einem speziellen, begrenzten Bereich auf der Festplatte des Benutzers speichern und lesen.

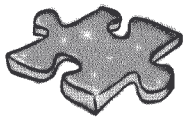
### Punkt für Punkt

- Die Java Web Start-Technik ermöglicht Ihnen das Deployment einer Standalone-Client-Anwendung über das Web.
- Zu Java Web Start (JWS) gehört eine »Hilfsanwendung«, die (neben Java) auf dem Client installiert sein muss.
- Ein JWS-Programm besteht aus zwei Teilen: einem ausführbaren JAR und einer .jnlp-Datei.
- Eine jnlp-Datei ist ein einfaches XML-Dokument, das Ihre JWS-Anwendung beschreibt. Es enthält Tags, in denen der Name und der Ort des JAR und der Name der Klasse mit der main()-Methode angegeben werden.
- Wenn ein Browser eine jnlp-Datei vom Server erhält (weil der Benutzer auf einen Link zur .jnlp-Datei geklickt hat), startet der Browser die JWS-Hilfsanwendung.
- Die JWS-Hilfsanwendung liest die jnlp-Datei und fordert das ausführbare JAR vom Webserver an.
- Wenn die JWS-Anwendung das JAR erhält, ruft sie die main()-Methode auf (die in der jnlp-Datei angegeben ist).

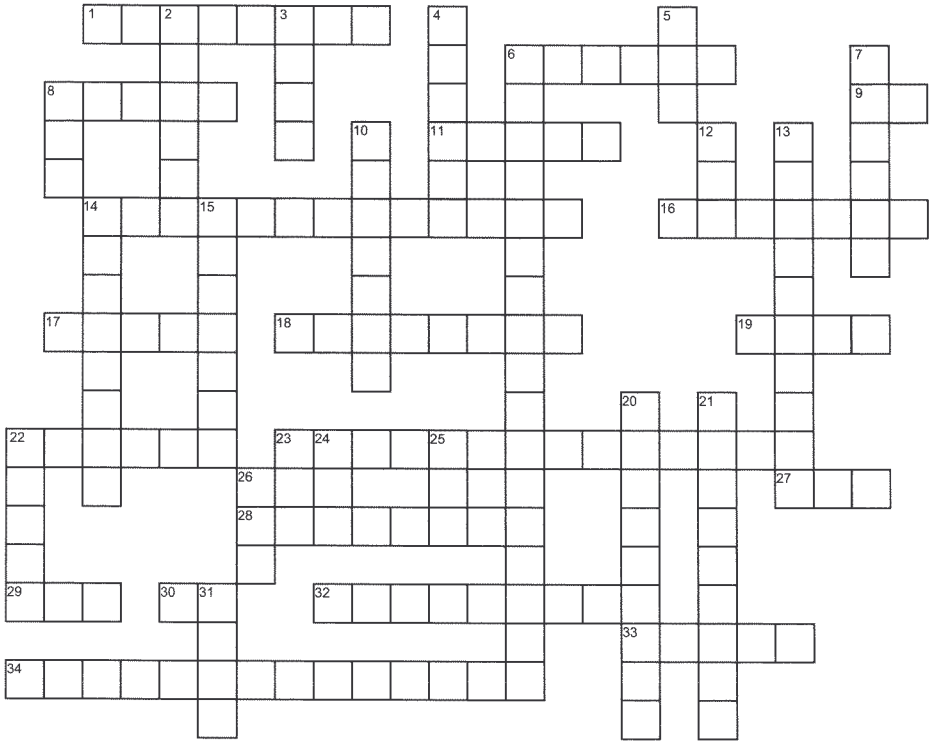
In diesem Kapitel haben wir das Packen von Paketen, das Deployment und JWS erkundet. Entscheiden Sie, ob die folgenden Aussagen jeweils wahr oder falsch sind.

# Wahr oder Falsch?

1) Der Java-Compiler hat einen Schalter, -d, mit dem Sie festlegen können, wo Ihre class-Dateien hinsollen.
2) Ein JAR ist ein normales Verzeichnis, in dem Ihre class-Dateien liegen sollten.
3) Wenn Sie ein Java-Archiv erzeugen, müssen Sie eine Datei namens jar.mf erstellen.
4) Die Hilfsdatei in einem Java-Archiv gibt an, in welcher Klasse sich die main()-Methode befindet.
5) JAR-Dateien müssen entpackt werden, bevor die JVM die Klassen darin benutzen kann.
6) Auf der Kommandozeile werden Java-Archive mit Hilfe des Schalters -arch aufgerufen.
7) Paketstrukturen werden sinnvollerweise mit Hilfe von Hierarchien dargestellt.
8) Von der Verwendung Ihres Firmen-Domainnamens bei der Bezeichnung von Paketen wird abgeraten.
9) Verschiedene Klassen innerhalb einer Quelldatei können zu verschiedenen Paketen gehören.
10) Es wird nachdrücklich geraten, beim Kompilieren von Klassen in einem Paket den Schalter -p zu verwenden.
11) Beim Kompilieren von Klassen in einem Paket muss der vollständige Name den Verzeichnisbaum widerspiegeln.
12) Eine sachgemäße Verwendung des Schalters -d kann sicherstellen, dass Ihr Klassenbaum frei von Tippfehlern ist.
13) Beim Extrahieren eines JAR mit Paketen wird ein Verzeichnis namens meta-inf erzeugt.
14) Beim Extrahieren eines JAR mit Paketen wird eine Datei namens manifest.mf erzeugt.
15) Die JWS-Hilfsanwendung läuft ausschließlich in Verbindung mit einem Browser.
16) JWS-Anwendungen benötigen eine .nlp-Datei (Network Launch Protocol), damit sie richtig funktionieren.
17) Die main-Methode eines JWS-Programms wird in seiner JAR-Datei angegeben.



# Kreuzworträtsel 7.0 ConTutto



Alles aus diesem Buch kann vorkommen!

## Waagerecht

1 Nur im Klassenbewusstsein.

6 Swing-Jargon.

8 E/A-Kanal.

9 Java-Verzweigung.

11 Milli-Nickerchen.

14 Flach drücken.

16 Javas Einwickelpapier.

17 Aufräumen bei E/A.

18 GUI-Zielperson.

19 Bitte acht Bit!

22 Effizienzexperte.

23 Verwirklichen.

27 Trigonometrie-Methode.

28 Wo soll ich starten.

29 Mutig sein.

30 Passiert paarweise.

32 Mini-Firewall.

33 Früh raus.

34 Nur mit Schlüssel!

## Senkrecht

2 Bitte nicht spalten!

3 Schön anordnen.

4 Objektbauplan.

5 Gibt Gekapseltes.

6 So gut wie neu.

7 Ja oder nein.

8 Kapselungsmethode.

10 Kein Verhalten.

12 Liefergefäß.

13 Verteilung und Einsatz.

14 Und noch mal hindurch ....

15 Häufiger Wrapper.

20 Aufdringliches Widget.

21 Nicht von bleibendem Wert.

22 Java-Team mit Hierarchie.

24 Abrunden.

25 Das Ende von VB.

26 JNLP-Format.

31 Wo ein Socket wohnt.

1. der Benutzer klickt auf einen Link auf einer Webseite
2. der Browser fordert eine .jnlp-Datei vom Webserver an
3. der Webserver sendet eine .jnlp-Datei an den Browser
4. der Webbrowser startet die JWS-Hilfsanwendung
5. die JWS-Hilfsanwendung fordert die JAR-Datei an
6. der Webserver sendet eine JAR-Datei an die JWS-Hilfsanwendung
7. die JWS-Hilfsanwendung ruft die main()-Methode des JAR auf

**Wahr** 1) Der Java-Compiler hat einen Schalter, -d, mit dem Sie festlegen können, wo Ihre class-Dateien hinsollen.

**Falsch** 2) Ein JAR ist ein normales Verzeichnis, in dem Ihre class-Dateien liegen sollten.

**Falsch** 3) Wenn Sie ein Java-Archiv erzeugen, müssen Sie eine Datei namens jar.mf erstellen.

**Wahr** 4) Die Hilfsdatei in einem Java-Archiv gibt an, in welcher Klasse sich die main()-Methode befindet.

**Falsch** 5) JAR-Dateien müssen entpackt werden, bevor die JVM die Klassen darin benutzen kann.

**Falsch** 6) Auf der Kommandozeile werden Java-Archive mit Hilfe des Schalters -arch aufgerufen.

**Wahr** 7) Paketstrukturen werden sinnvollerweise mit Hilfe von Hierarchien dargestellt.

**Falsch** 8) Von der Verwendung Ihres Firmen-Domainnamens bei der Bezeichnung von Paketen wird abgeraten.

**Falsch** 9) Verschiedene Klassen innerhalb einer Quelldatei können zu verschiedenen Paketen gehören.

**Falsch** 10) Es wird nachdrücklich geraten, beim Kompilieren von Klassen in einem Paket den Schalter -p zu verwenden.

**Wahr** 11) Beim Kompilieren von Klassen in einem Paket muss der vollständige Name den Verzeichnisbaum widerspiegeln.

**Wahr** 12) Eine sachgemäße Verwendung des Schalters -d kann sicherstellen, dass Ihr Klassenbaum frei von Tippfehlern ist.

**Wahr** 13) Beim Extrahieren eines JAR mit Paketen wird ein Verzeichnis namens meta-inf erzeugt.

**Wahr** 14) Beim Extrahieren eines JAR mit Paketen wird eine Datei namens manifest.mf erzeugt.

**Falsch** 15) Die JWS-Hilfsanwendung läuft ausschließlich in Verbindung mit einem Browser.

**Falsch** 16) JWS-Anwendungen benötigen eine nlp-Datei (Network Launch Protocol), damit sie richtig funktionieren.

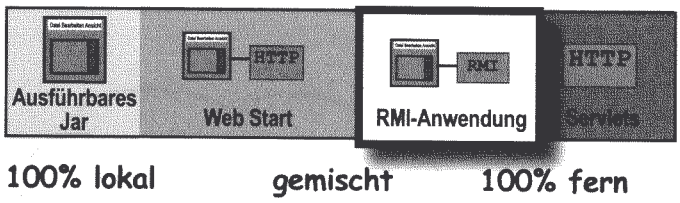
**Falsch** 17) Die main-Methode eines JWS-Programms wird in seiner JAR-Datei angegeben.

# Lösung zum Kreuzworträtsel 7.0 ConTutto

18 Ferndeployment mit RMI

# Verteilte Anwendungen

**Entfernung muss kein Problem sein.** Sicher, es *ist* einfacher, wenn alle Teile Ihrer Anwendung an einem Ort sind, auf einem Heap, mit einer JVM, die über alles herrscht. Aber das ist nicht immer möglich. Und auch nicht immer wünschenswert. Was ist, wenn Ihr Programm leistungsintensive Berechnungen durchführt, aber die Endanwender an einem mickrigen Java-fähigen Kleingerät sitzen? Was ist, wenn Ihr Programm Daten aus einer Datenbank braucht, aber aus Sicherheitsgründen nur Code von Ihrem Server auf die Datenbank zugreifen darf? Stellen Sie sich z.B. ein großes E-Commerce-Backend vor, das innerhalb eines Transaktionsmanagement-Systems laufen muss. Manchmal *muss* ein Teil Ihres Programms auf einem Server laufen, während ein anderer Teil (üblicherweise der Client) auf einer *anderen* Maschine laufen muss. In diesem Kapitel lernen wir, wie man die erstaunlich einfache Java-Technik namens Remote Method Invocation (RMI) einsetzt. Wir werfen auch einen kurzen Blick auf Servlets, Enterprise Java Beans (EJB) und Jini und sehen uns an, in welcher Weise EJB und Jini von RMI *abhängen*. Am Schluss des Buchs schreiben wir dann etwas, das zu den coolsten Sachen gehört, die man mit Java überhaupt machen kann: einen *Universaldienstbrowser*.

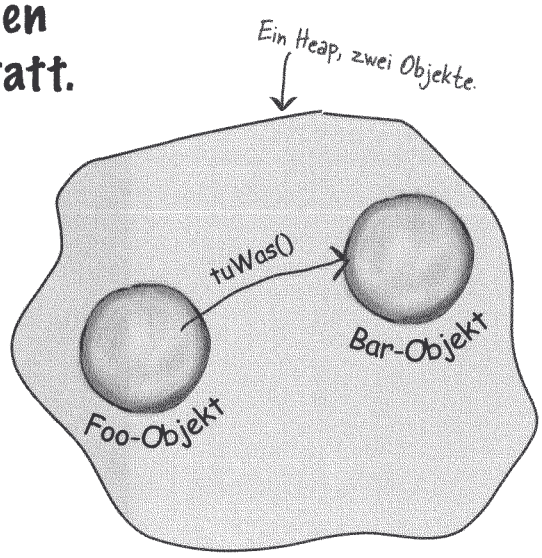



# Methodenaufrufe finden immer zwischen zwei Objekten auf dem gleichen Heap statt.

Alle Methoden, die wir bisher in diesem Buch aufgerufen haben, gehörten zu einem Objekt, das innerhalb der gleichen virtuellen Maschine ausgeführt wurde wie der Aufrufer. Mit anderen Worten, das aufrufende und das aufgerufene Objekt (dessen Methode aufgerufen wird) befinden sich auf dem gleichen Heap.

```
class Foo {
  void los() {
    Bar b = new Bar();
    b.tuWas();
  }
  public static void main(String[] args) {
    Foo f = new Foo();
    f.los();
  }
}
```

Im obigen Code ist klar, dass sich die von *f* referenzierte Foo-Instanz und das von *b* referenzierte Bar-Objekt auf dem gleichen Heap befinden und von der gleichen JVM ausgeführt werden. Erinnern Sie sich: Die JVM ist dafür verantwortlich, Bits in die Referenzvariable zu schreiben, die repräsentiert, wie man *an ein Objekt auf dem Heap gelangt*. Die JVM weiß jederzeit, wo sich die einzelnen Objekte befinden und wie man an sie herankommt. Aber die JVM kann ausschließlich Referenzen auf ihren *eigenen* Heap kennen! Eine JVM , die auf *einem* Rechner läuft, kann beispielsweise nicht über den Heap-Raum einer JVM Bescheid wissen, die auf einem *anderen* Rechner läuft. Tatsächlich kann eine JVM, die auf einem Rechner läuft, noch nicht einmal etwas über eine andere JVM wissen, die auf dem *gleichen* Rechner läuft. Es macht keinen Unterschied, ob die JVMs auf dem gleichen oder auf verschiedenen Rechnern laufen. Entscheidend ist: Die beiden JVMs sind ... na ja, eben zwei unterschiedliche JVM-Aufrufe.



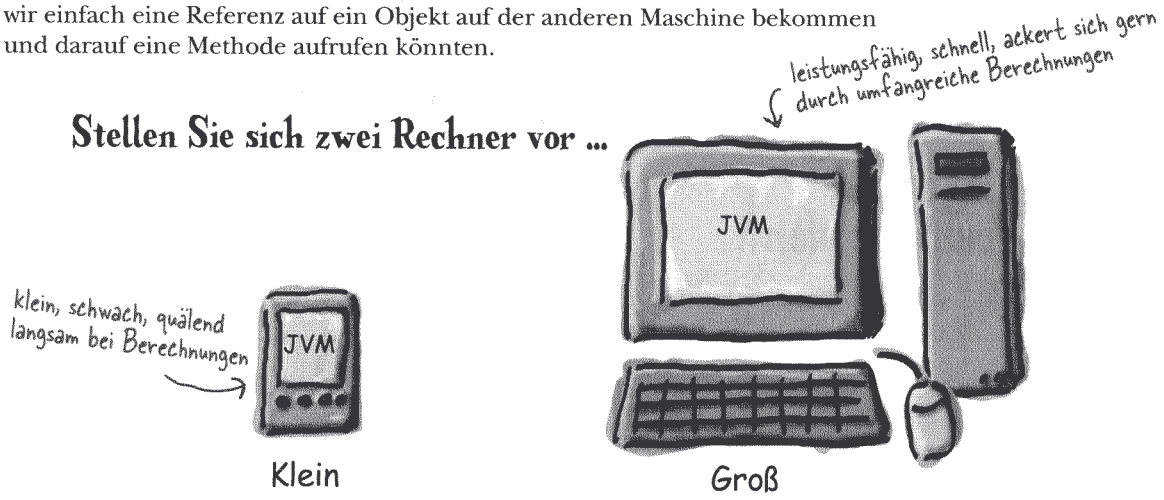

Wenn ein Objekt eine Methode auf einem anderen aufruft, gehören diese Objekte bei den meisten Anwendungen zum gleichen Heap. Anders ausgedrückt, beide werden innerhalb der gleichen JVM ausgeführt.

# Und wenn Sie nun eine Methode auf einem Objekt aufrufen wollen, das auf einem anderen Rechner läuft?

Wir wissen, wie wir Informationen von einer Maschine auf eine andere übertragen – mit Sockets und E/A. Wir öffnen eine Socket-Verbindung zu einer anderen Maschine, besorgen uns einen Ausgabestrom und schreiben Daten hinein.

Wenn wir aber nun eine *Methode auf etwas aufrufen* wollen, das auf einer anderen Maschine läuft ... einer anderen JVM? Wir könnten dann natürlich unser eigenes Protokoll erstellen, und wenn wir Daten an einen ServerSocket senden, könnte der Server sie analysieren, herausfinden, was wir gemeint haben, die Arbeit erledigen und das Ergebnis über einen anderen Strom zurücksenden. Aber wie mühsam wäre das! Stellen Sie sich vor, wie viel schöner es wäre, wenn wir einfach eine Referenz auf ein Objekt auf der anderen Maschine bekommen und darauf eine Methode aufrufen könnten.

**Stellen Sie sich zwei Rechner vor ...**




**Groß hat etwas, was Klein haben möchte.**
**Rechenleistung.**

**Klein möchte Daten an Groß senden, damit Groß die Schwerstarbeit erledigen kann.**

**Klein möchte einfach eine Methode aufrufen ...**

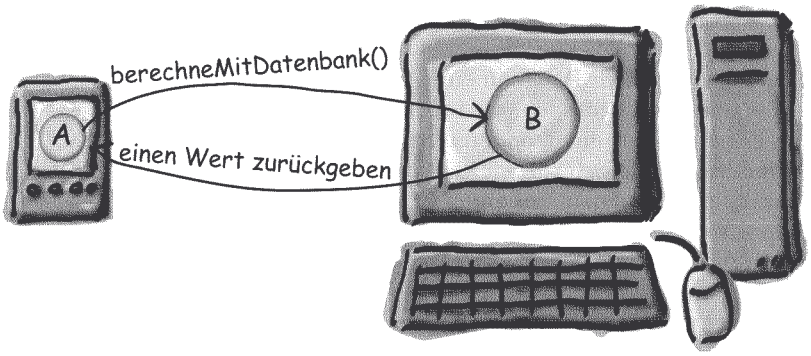
```
double berechneMitDatenbank(RechenZahlen zahlen)
```

**... und das Ergebnis zurückbekommen.**

**Aber wie kommt Klein an eine Referenz auf ein Objekt von Groß?**

## Objekt A, das auf Klein läuft, möchte eine Methode auf Objekt B aufrufen, das auf Groß läuft.

Die Frage ist: Wie bekommen wir es hin, dass ein Objekt auf einer Maschine eine Methode auf einer anderen Maschine (d.h. einem anderen Heap oder einer anderen JVM) aufruft?




## Aber das geht doch gar nicht.

Na ja, nicht direkt jedenfalls. Sie können keine Referenz auf etwas auf einem anderen Heap bekommen. Wenn Sie sagen:

Hund h = ???

... dann muss das, was durch *h* referenziert wird, im gleichen Heap-Raum liegen wie der die Anweisung ausführende Code.

Aber stellen Sie sich vor, Sie wollen etwas entwerfen, das Ihre Absicht (einen Methodenaufruf auf einer anderen Maschine) mit Hilfe von Sockets und E/A übermittelt und dennoch für Sie wie ein lokaler Methodenaufruf aussieht.

Mit anderen Worten, Sie wollen einen Methodenaufruf auf einem *entfernten* Objekt bewirken (d.h. auf einem Objekt in einem Heap »irgendwo anders«), aber mit Code, mit dem Sie *so tun können*, als riefen Sie eine Methode auf einem lokalen Objekt auf. So einfach wie ein guter alter alltäglicher Methodenaufruf – aber mit der Leistungsfähigkeit des Fernaufrufs von Methoden. Das ist unser Ziel.

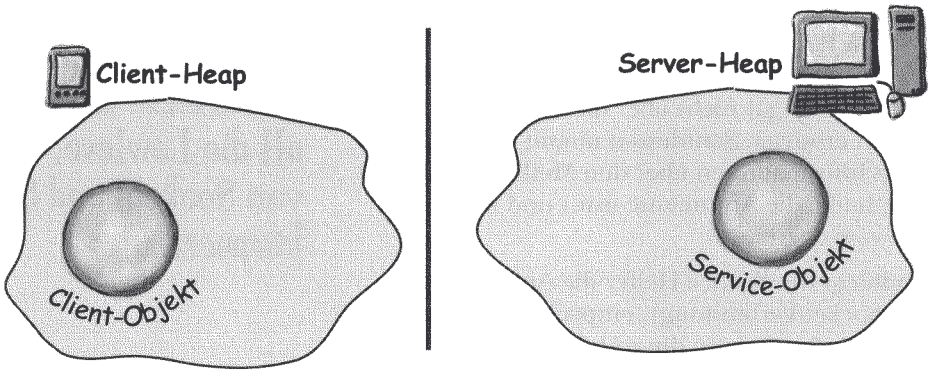
Und genau das bietet Ihnen RMI (Remote Method Invocation)!

Doch gehen wir noch einmal einen Schritt zurück und stellen uns vor, wie wir RMI konzipieren würden, wollten wir es selbst entwickeln. Wenn Sie verstehen, was Sie selbst entwickeln müssten, werden Sie auch besser verstehen, wie RMI funktioniert.

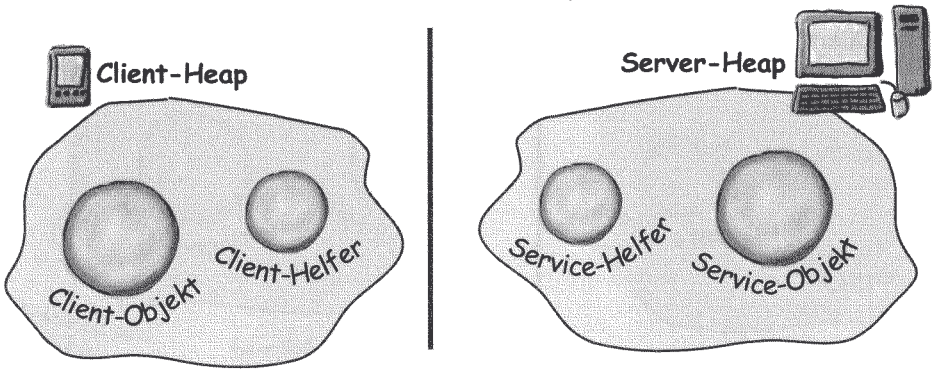
# Ein Entwurf für den Fernaufruf von Methoden

## Erzeugen Sie vier Dinge: Server, Client, Server-Helfer, Client-Helfer

1. Erzeugen Sie Client- und Server-Anwendungen. Die Server-Anwendung ist der **Remote-Service**, ein »Ferndienst«, der ein Objekt mit der Methode besitzt, die der Client aufrufen will.




2. Erzeugen Sie Client- und Server-»Helfer«. Diese kümmern sich um all die Lowlevel-Netzwerk- und E/A-Details, mit deren Hilfe Ihr Client und Ihr Service (den wir auch »Dienst« nennen könnten) so tun können, als gehörten sie zum selben Heap.



# Die Rolle der »Helfer«

Die »Helfer« sind die Objekte, die sich um die Kommunikation kümmern. Sie ermöglichen es dem Client, sich so zu verhalten, als riefe er eine Methode auf einem lokalen Objekt auf (was er tatsächlich auch tut). Der Client ruft eine Methode auf dem Client-Helfer auf, *als wäre der Client-Helfer der eigentliche Service. Der Client-Helfer ist ein Proxy, eine Art Stellvertreter oder Bevollmächtigter für den echten Service.*

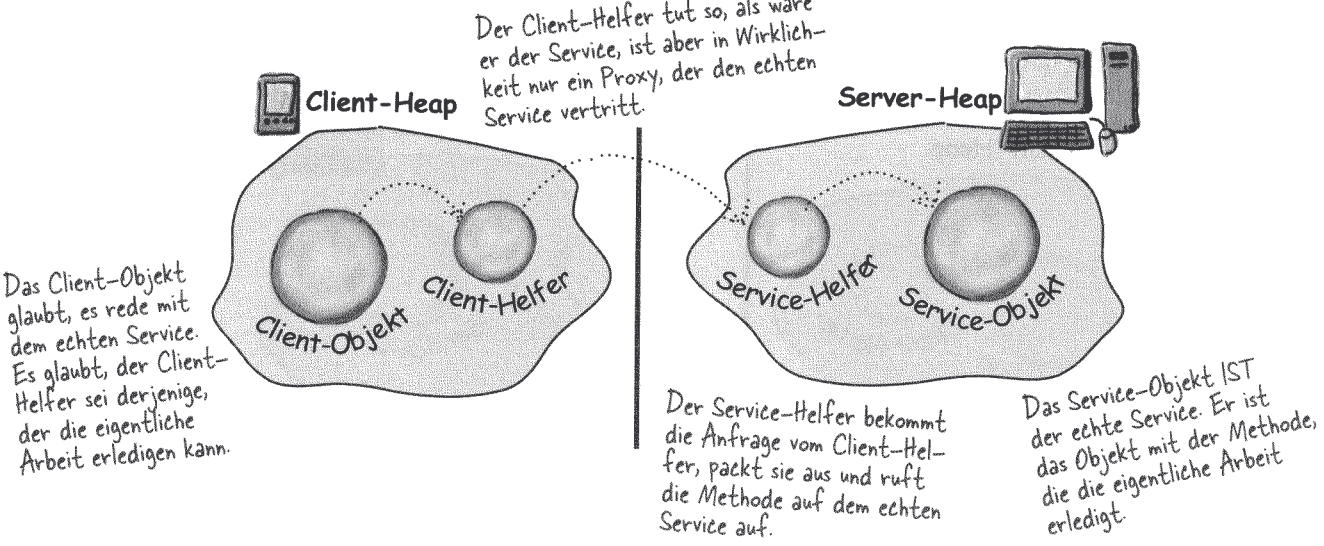
Anders ausgedrückt: Das Client-Objekt *glaubt*, es rufe eine Methode auf dem Remote-Service auf. Denn der Client-Helfer *täuscht vor*, er selbst wäre das Service-Objekt. ***Der tut doch glatt so, als besäße er selbst die Methode, die der Client aufrufen will!***

Aber der Client-Helfer ist *in Wirklichkeit nicht* der Remote-Service. Er *benimmt* sich zwar so (weil er dieselbe Methode besitzt, die auch der Service anbietet), aber er verfügt gar nicht über die eigentliche Methodenlogik, die der Client erwartet. Stattdessen nimmt er Kontakt zum Server auf, überträgt Informationen über den Methodenaufruf (z.B. den Namen der Methode, die Argumente usw.) und wartet dann auf eine Rückgabe vom Server.

Auf der Server-Seite empfängt der Service-Helfer die Anfrage vom Client-Helfer (über eine Socket-Verbindung), entpackt die Informationen über den Aufruf und ruft dann die *echte* Methode auf dem *echten* Service-Objekt auf. Für das Service-Objekt ist das also ein lokaler Aufruf, denn er kommt vom Service-Helfer, nicht von einem entfernten Client.

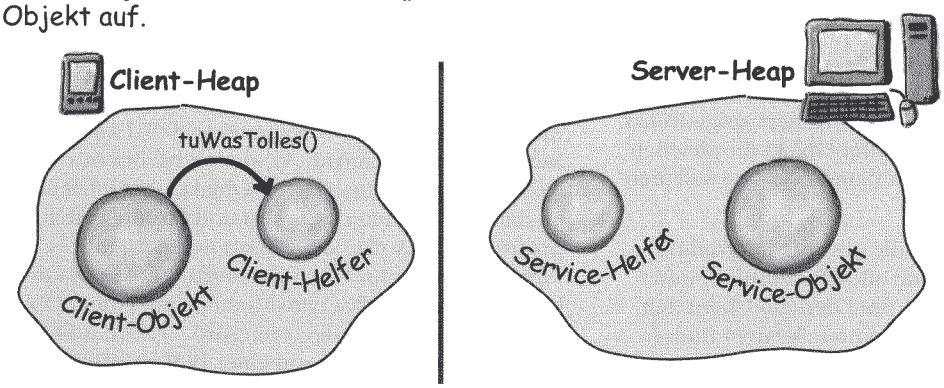
Der Service-Helfer erhält den Rückgabewert vom Service, verpackt ihn und schickt ihn zurück an den Client-Helfer (über einen Socket-Ausgabestrom). Der Client-Helfer packt die Informationen aus und gibt den Wert an das Client-Objekt weiter.

**Ihr Client-Objekt handelt, als ob es Fernaufrufe von Methoden ausführt. Aber in Wirklichkeit tut es nichts weiter, als Methoden auf einem »Proxy«-Objekt im lokalen Heap aufzurufen, das sich um all die Lowlevel-Details von Sockets und Strömen kümmert.**

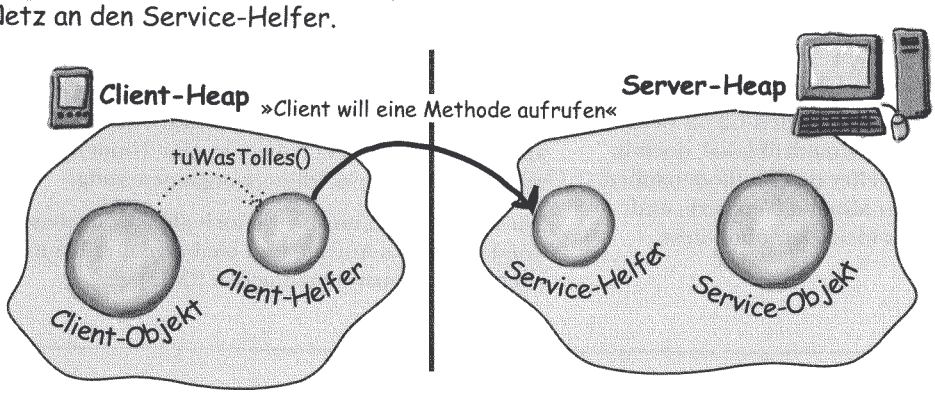


# Was beim Aufruf der Methode passiert

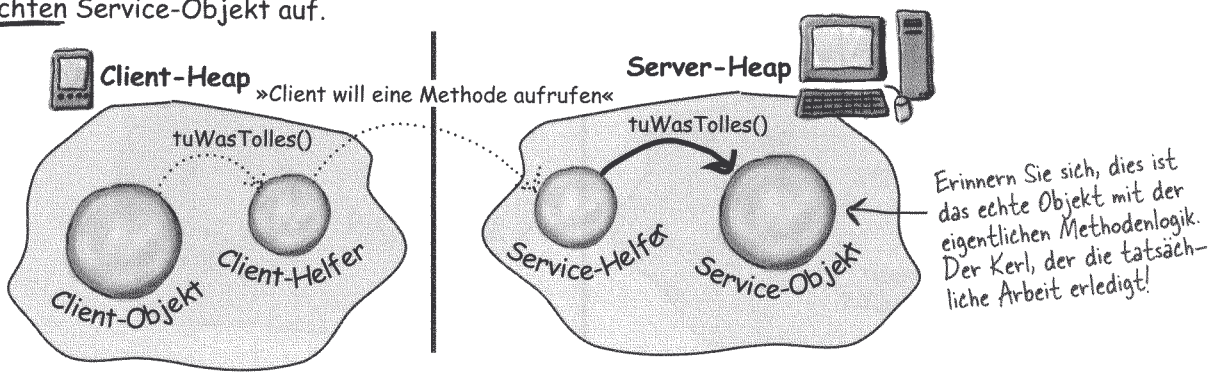
1. Client-Objekt ruft tuWasTolles() auf dem Client-Helfer-Objekt auf.




2. Client-Helfer verpackt die Information über den Aufruf (Argumente, Methodenname usw.) und schickt sie über das Netz an den Service-Helfer.




3. Service-Helfer packt Information vom Client-Helfer aus, findet heraus, welche Methode aufzurufen ist (und auf welchem Objekt) und ruft die echte Methode auf dem echten Service-Objekt auf.



# Java-RMI gibt Ihnen die Client- und Service-Helfer-Objekte

In Java erstellt RMI die Client- und Service-Helfer-Objekte für Sie, und es weiß sogar, wie man den Client-Helfer aussehen lässt wie den echten Service. Anders ausgedrückt: RMI weiß, wie man dem Client-Helfer-Objekt die gleichen Methoden gibt, die man auf dem Remote-Service aufrufen will.

Und außerdem stellt RMI auch die gesamte Laufzeitinfrastruktur zur Verfügung, damit alles funktioniert, unter anderem einen Lookup-Service, über den der Client den Client-Helfer (den Proxy für den echten Service) finden kann.

Mit RMI müssen Sie nichts von dem Netzwerk- oder E/A-Code selbst schreiben. Ihr Client kann Remote-Methoden (d.h. die Methoden des echten Service) aufrufen, als wären es normale Methodenaufrufe auf Objekten, die in seiner eigenen lokalen JVM ausgeführt werden.

Jedenfalls fast.

*Ein* Unterschied besteht zwischen RMI-Aufrufen und lokalen (normalen) Methodenaufrufen: Für den Client sieht es zwar so aus, als wäre der Methodenaufruf lokal, doch in Wirklichkeit sendet der Client-Helfer den Methodenaufruf über das Netz. Wir haben es hier also mit Netzwerk- und E/A-Methoden zu tun. Und was wissen wir über diese Methoden?

Sie sind riskant!

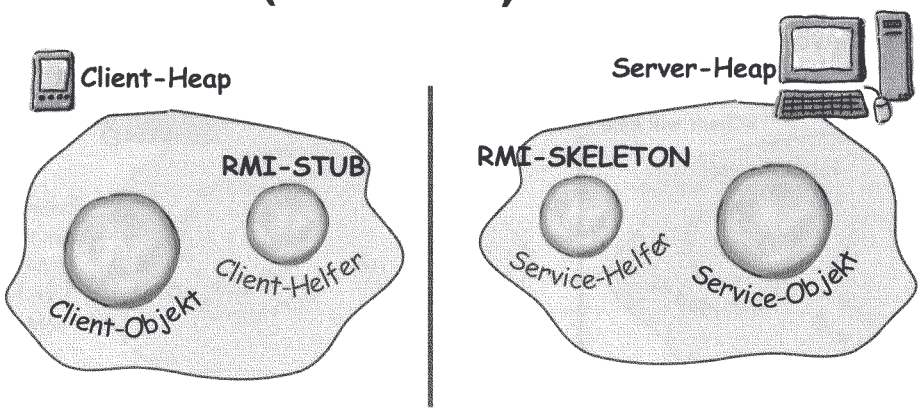
Sie werfen mit Exceptions nur so um sich!

Daher muss der Client dieses Risiko einkalkulieren. Er muss bestätigen, dass er Folgenes weiß: Wenn er eine Remote-Methode aufruft, ist es zwar für ihn nur ein lokaler Aufruf auf dem Proxy/Helfer-Objekt, aber letztendlich sind an diesem Aufruf auch Sockets und Ströme beteiligt. Der ursprüngliche Aufruf des Clients ist zwar *lokal*, aber der Proxy macht daraus einen Fernaufruf. Ein Fernaufruf – das bedeutet einfach, dass eine Methode auf einem Objekt in einer anderen JVM aufgerufen wird. *Wie* die Informationen über diesen Aufruf von einer JVM zu einer anderen übertragen werden, hängt vom Protokoll ab, das die Helfer-Objekte verwenden.

Mit RMI können Sie das Protokoll auswählen: JRMP oder IIOP. JRMP ist das »native« Protokoll von RMI, also das Protokoll, das speziell für Von-Java-zu-Java-Fernaufrufe gemacht wurde. IIOP ist das Protokoll für CORBA (Common Object Request Broker Architecture); damit sind auch Fernaufrufe auf Nicht-Java-Objekten möglich. CORBA ist normalerweise *viel* mühsamer als RMI, denn wenn Sie nicht auf beiden Seiten Java haben, ist eine Unmenge an Übersetzungs- und Umwandlungsvorgängen nötig.

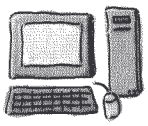
Wir interessieren uns hier jedoch glücklicherweise nur für Von-Java-zu-Java-Aufrufe und bleiben daher beim guten, alten, bemerkenswert einfachen RMI.

## In RMI heißt der Client-Helfer »Stub« (deutsch »Stumpf«) und der Service-Helfer »Skeleton« (»Skelett«).



# Die Erstellung des Remote-Service

Dies ist ein **Überblick** über die fünf Schritte zum Aufbau eines Remote-Service (der auf dem Server läuft). Seien Sie unbesorgt, jeder Schritt wird auf den folgenden Seiten ganz genau erklärt.




## Schritt 1:

### Remote-Interface erstellen

Das Remote-Interface definiert die Methoden, die ein Client aus der Ferne aufrufen kann. Es wird vom Client als polymorpher Typ für den Service benutzt werden. Die Methoden werden sowohl vom Stub als auch vom echten Service implementiert!

MeinRemote.java

Dieses Interface definiert die Remote-Methoden, die von Clients aufgerufen werden sollen.

## Schritt 2:

### Remote-Implementierung erstellen

Diese Klasse leistet die eigentliche Arbeit. In ihr sind die Remote-Methoden, die im Remote-Interface definiert sind, wirklich implementiert. Dies ist das Objekt, auf dem der Client eigentlich Methoden aufrufen möchte.

MeinRemoteImpl.java

Der echte Service, d.h. die Klasse mit den Methoden, die die eigentliche Arbeit verrichten. Sie implementiert das Remote-Interface.

## Schritt 3:

### Mit rmic Stubs und Skeletons erzeugen

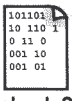
Dies sind die Client- und Service-»Helfer«. Diese Klassen müssen Sie nicht selbst erzeugen; auch den Quellcode für die Erzeugung brauchen Sie sich nicht anzusehen. Das passiert alles automatisch, wenn Sie das rmic-Werkzeug verwenden, das zu Ihrem Java Development Kit gehört.

Wir führen rmic für die echte Service-Implementierungsklasse aus ...

```
Datei Bearbeiten Fenster Hilfe Essen
%rmic MeinRemoteImpl
```

... und es spuckt uns zwei neue Klassen für die Helfer-Objekte aus.

MeinRemoteImpl_Stub.class




MeinRemoteImpl_Skel.class

## Schritt 4:

### RMI-Registrierung (rmiregistry) starten

Die *RMI-Registrierung* ähnelt einem Telefonbuch. Hier findet der Client den Proxy (das Client-Stub/Helfer-Objekt).

```
Datei Bearbeiten Fenster Hilfe Trinken
%rmiregistry
```

Führen Sie dies in einem eigenen Kommandozeilen-Fenster aus.

## Schritt 5:

### Remote-Service starten

Jetzt müssen Sie nur noch das Service-Objekt zum Einsatz bringen. Ihre Service-Implementierungsklasse instantiiert eine Service-Instanz und meldet sie bei der RMI-Registrierung an. Durch diese Registrierung wird der Service für Clients verfügbar.

```
Datei Bearbeiten Fenster Hilfe LustigSein
%java MeinRemoteImpl
```

# Schritt 1: Remote-Interface erstellen

MeinRemote.java

### ① java.rmi.Remote erweitern

Remote ist ein »Marker«-Interface, das zwar keine Methoden enthält, jedoch im Rahmen von RMI eine besondere Rolle spielt; daher muss diese Regel befolgt werden. Beachten Sie auch, dass wir hier »extends« schreiben: Ein Interface darf ein anderes Interface *erweitern*.

```
public interface MeinRemote extends Remote {
```

Ihr Interface muss verkünden, dass es zum Aufruf von Remote-Methoden dient. Ein Interface kann nichts implementieren, aber es kann andere Interfaces erweitern.

### ② Für alle Methoden das Auslösen einer Remote-Exception deklarieren

Das Remote-Interface wird vom Client als polymorpher Typ für den Service eingesetzt. Mit anderen Worten: Der Client ruft Methoden auf etwas auf, das das Remote-Interface implementiert. Dieses »Etwas« ist unser Stub, und weil zu dessen Geschäft Netzwerk und E/A gehören, kann alles Mögliche schief gehen. Diese Risiken muss der Client berücksichtigen, indem er die Remote-Exceptions entweder behandelt oder weitergibt. Wenn die Methoden in einem Interface Exceptions deklarieren, muss jeder Code, der Methoden auf einer Referenz dieses Typs (d.h. des Interface-Typs) aufruft, diese Exceptions behandeln oder deklarieren.

```
import java.rmi.*;
public interface MeinRemote extends Remote {
  public String sagHallo() throws RemoteException;
}
```

Das Interface Remote gehört zu java.rmi.

Jeder Fernaufruf von Methoden ist als »riskant« einzustufen. Das Deklarieren der RemoteException in jeder Methode zwingt den Client, aufzupassen und einzukalkulieren, dass etwas nicht klappen könnte.

### ③ Argumente und Rückgabewerte müssen primitive Datenypen oder serialisierbar sein

Die Argumente und Rückgabewerte einer Remote-Methode müssen entweder primitive Typen oder serialisierbar sein. Überlegen Sie: Jedes Argument, das einer Remote-Methode übergeben wird, muss verpackt und über das Netzwerk versendet werden. Dafür benötigt man Serialisierung. Das Gleiche gilt für Rückgabewerte. Mit primitiven Datentypen, Strings und den meisten anderen API-Datentypen (einschließlich Arrays und Collections) werden Sie keine Probleme haben. Aber wenn Sie Ihre eigenen Typen durch die Gegend schicken, müssen Sie dafür sorgen, dass Ihre Klassen Serializable implementieren.

```
public String sagHallo() throws RemoteException;
```

Dieser Rückgabewert wird vom Server über die Leitung zurück an den Client geschickt und muss deshalb serialisierbar sein. Alle Argumente und Rückgabewerte werden auf diese Weise verpackt und versendet.

# Schritt 2: Remote-Implementierung erstellen

MeinRemoteImpl.java

### ① Das Remote-Interface implementieren

Ihr Service muss das Remote-Interface implementieren, d.h. das Interface, dessen Methoden Ihr Client aufrufen wird.

```
public class MeinRemoteImpl extends UnicastRemoteObject implements MeinRemote {
  public String sagHallo() {
    return "Server sagt Hallo";
  }
  // weiterer Code für diese Klasse
}
```

Der Compiler wird überprüfen, ob Sie alle Methoden implementiert haben, die das implementierte Interface enthält. In diesem Fall ist es nur eine.

### ② UnicastRemoteObject erweitern

Um als Remote-Service-Objekt arbeiten zu können, braucht Ihr Objekt Eigenschaften, die mit seinem »Dasein als entferntes Objekt« zusammenhängen. Am einfachsten ist es, UnicastRemoteObject (aus dem java.rmi.server-Paket) zu erweitern und diese Klasse (Ihre Superklasse) die Arbeit machen zu lassen.

```
public class MeinRemoteImpl extends UnicastRemoteObject implements MeinRemote {
```

### ③ Einen parameterlosen Konstruktor schreiben, der eine RemoteException deklariert

Mit der neuen Superklasse, UnicastRemoteObject, gibt es ein kleines Problem: Ihr Konstruktor löst eine RemoteException aus. Es gibt nur eine einzige Möglichkeit, mit diesem Problem fertig zu werden – schreiben Sie einen Konstruktor für Ihre Remote-Implementierung, in diesem können Sie dann die RemoteException deklarieren. Sie erinnern sich sicher, dass beim Instantiieren einer Klasse immer auch der Konstruktor der Superklasse aufgerufen wird. Und wenn der Konstruktor Ihrer Superklasse eine Exception auslöst, bleibt Ihnen nicht anderes übrig, als auch Ihren eigenen Konstruktor eine Exception auslösen zu lassen.

```
public MyRemoteImpl() throws RemoteException { }
```

Der Konstruktor darf leer sein. Sie brauchen lediglich eine Möglichkeit, die Exception zu deklarieren, die der Konstruktor Ihrer Superklasse auslöst.

### ④ Den Service bei der RMI-Registrierung anmelden

Jetzt müssen Sie Ihren Remote-Service noch für Remote-Clients verfügbar machen. Dafür instantiieren Sie ihn und tragen ihn in die RMI-Registrierung ein (diese muss vorher gestartet werden, andernfalls schlägt die Registrierung fehl). Wenn Sie Ihr Implementierungsobjekt registrieren, trägt das RMI-System in Wirklichkeit den *Stub* in die Registrierung ein, denn genau genommen ist es dieser, den der Client benötigt. Registrieren Sie Ihren Service mit der statischen Methode rebind() aus der Klasse java.rmi.Naming.

```
try {
  MeinRemote service = new MeinRemoteImpl();
  Naming.rebind("RemoteHallo", service);
} catch(Exception ex) {...}
```

Geben Sie Ihrem Service einen Namen (über den Clients ihn in der Registrierung nachschlagen können) und melden Sie ihn bei der RMI-Registrierung an. Wenn Sie das Service-Objekt anmelden, registriert RMI nicht den Service selbst, sondern den zugehörigen Stub.

# Schritt 3: Stubs und Skeletons erzeugen

### 1 rmic für die Remote-Implementierungsklasse ausführen (nicht für das Remote-Interface)

Wenn man dem rmic-Werkzeug (das Bestandteil des Java Software Development Kit ist) eine Service-Implementierung übergibt, erzeugt es zwei neue Klassen: Stub und Skeleton. Dabei verwendet es folgende Namenskonvention: An den Namen Ihrer Remote-Implementierung hängt es _Stub bzw. _Skel an. Es gibt eine Reihe anderer Optionen für rmic, z.B. die Ausführung ohne Erzeugung von Skeletons, mit Erzeugung von Quellcode-Dateien für diese Klassen oder gar unter Verwendung von IIOP als Protokoll. Wir benutzen rmic hier jedoch nur auf die übliche Weise. Die Klassen landen im aktuellen Verzeichnis (in das Sie mit `cd` gelangt sind). Denken Sie daran, dass *rmic* Ihre Implementierungsklasse einsehen können muss; vermutlich werden Sie rmic deshalb aus dem Verzeichnis starten, in dem Ihre Remote-Implementierung liegt. (Um es einfacher zu machen, verwenden wir hier absichtlich keine Pakete. In der echten Welt müssen Sie Paket-Verzeichnisstrukturen und vollständige Namen benutzen.)

Beachten Sie: Schreiben Sie nicht >>.class<< ans Ende, sondern nur den Klassennamen.

```
Datei Bearbeiten Fenster Hilfe Essen
%rmic MeinRemoteImpl
```

rmic erzeugt zwei neue Klassen für die Helfer-Objekte.

MeinRemoteImpl_Stub.class

MeinRemoteImpl_Skel.class

# Schritt 4: rmiregistry ausführen

### 1 Ein Kommandozeilen-Fenster öffnen und rmiregistry starten

Sorgen Sie dafür, dass Sie in einem Verzeichnis sind, aus dem auf Ihre Klassen zugegriffen werden kann. Am einfachsten ist es, wenn Sie rmiregistry direkt aus Ihrem Verzeichnis »classes« starten.

```
Datei Bearbeiten Fenster Hilfe Häh?
%rmiregistry
```

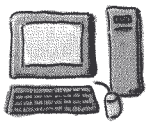
# Schritt 5: Service starten

### 1 Noch ein Kommandozeilen-Fenster öffnen und den Service starten

Dies kann aus einer main()-Methode in Ihrer Remote-Implementierungsklasse heraus erfolgen oder über eine spezielle Starterklasse. In diesem einfachen Beispiel stecken wir den Startcode in die Implementierungsklasse in eine main()-Methode, die das Objekt instantiiert und es bei der RMI-Registrierung anmeldet.

```
Datei Bearbeiten Fenster Hilfe Häh?
%java MeinRemoteImpl
```

# Der vollständige Code für die Server-Seite




## Das Remote-Interface:

```
import java.rmi.*;

public interface MeinRemote extends Remote {

  public String sagHallo() throws RemoteException;
}
```

RemoteException und Interface Remote gehören zum java.rmi-Paket.

Ihr Interface MUSS java.rmi.Remote erweitern.

Jede einzelne Ihrer Remote-Methoden muss eine RemoteException deklarieren.

---

## Der Remote-Service (die Implementierung):

```
import java.rmi.*;
import java.rmi.server.*;

public class MeinRemoteImpl extends UnicastRemoteObject implements MeinRemote {

  public String sagHallo() {
    return "Server sagt Hallo";
  }

  public MeinRemoteImpl() throws RemoteException { }

  public static void main (String[] args) {

    try {
      MeinRemote service = new MeinRemoteImpl();
      Naming.rebind("RemoteHallo", service);
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

UnicastRemoteObject gehört zum Paket java.rmi.server.

Die Erweiterung von UnicastRemoteObject ist der einfachste Weg, ein Remote-Objekt zu erzeugen.

Sie MÜSSEN Ihr Remote-Interface implementieren!!

Natürlich müssen Sie alle Interface-Methoden implementieren. Aber beachten Sie, dass die RemoteException hier NICHT deklariert werden muss.

Ihr Superklassen-Konstruktor (für UnicastRemoteObject) deklariert eine Exception. SIE müssen daher hier einen Konstruktor schreiben, denn das bedeutet, dass Ihr Konstruktor riskanten Code (d.h. seinen Super-Konstruktor) aufruft.

Erzeugen Sie das Remote-Objekt und >>binden<< Sie es dann mit der statischen Methode Naming.rebind() an die RMI-Registrierung. Der Name, mit dem Sie es anmelden, ist auch der Name, unter dem ein Client es in der RMI-Registrierung nachschlägt.

# Wie gelangt der Client an das Stub-Objekt?

Der Client muss an das Stub-Objekt kommen, denn genau darauf wird der Client Methoden aufrufen. Und da kommt die RMI-Registrierung ins Spiel. Der Client schlägt nach – wie in einem Telefonbuch – und sagt schließlich: »Ich hätte gern den Stub, der zu diesem Namen hier gehört.«

lookup() ist eine statische Methode der Klasse Naming.

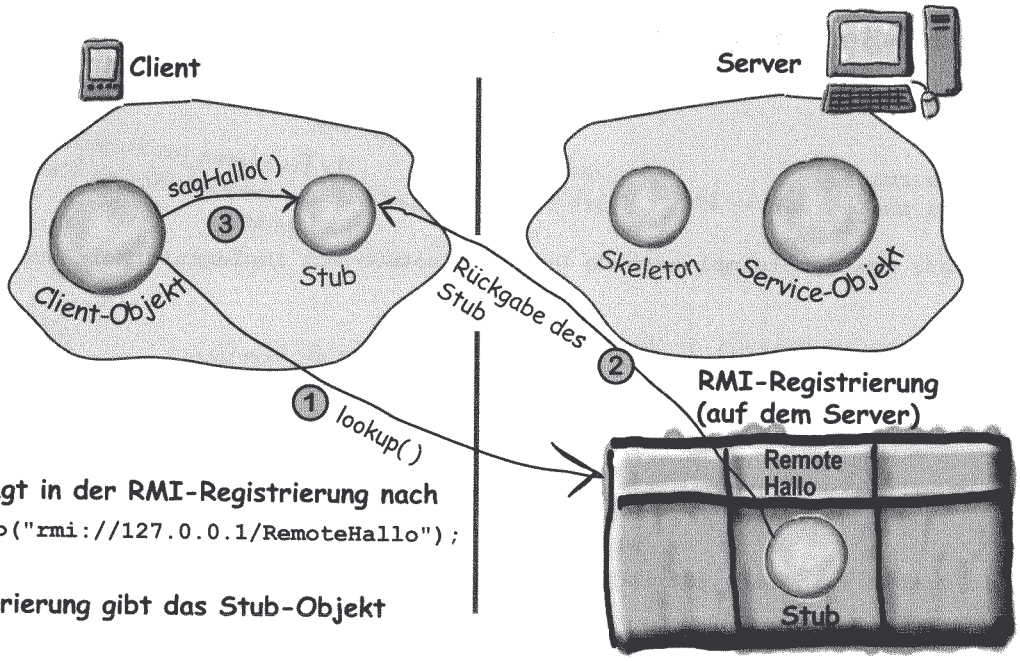
Dies muss der Name sein, unter dem der Service angemeldet wurde.

```
MeinRemote service = (MeinRemote) Naming.lookup("rmi://127.0.0.1/RemoteHallo");
```

Als Typ für den Service verwendet der Client immer das Remote-Interface. Den Namen Ihrer Remote-Service-Klasse muss der Client gar nicht wissen.

Es ist eine Umwandlung in den Interface-Typ nötig, weil die lookup()-Methode den Datentyp Object zurückgibt.

Der Hostname oder die IP-Adresse.




**1 Client schlägt in der RMI-Registrierung nach**

```
Naming.lookup("rmi://127.0.0.1/RemoteHallo");
```

**2 RMI-Registrierung gibt das Stub-Objekt zurück**

(als Rückgabewert der lookup-Methode), und RMI deserialisiert automatisch den Stub. Die Stub-Klasse (die rmic für Sie erzeugt hat) MUSS auf dem Client-Rechner vorhanden sein, sonst wird der Stub nicht deserialisiert.

**3 Client ruft eine Methode auf dem Stub auf, als wäre der Stub WIRKLICH der Service**

# Wie kommt der Client an die Stub-Klasse?

Jetzt kommen wir zu einer interessanten Frage: Wie kriegen wir es hin, dass die Stub-Klasse (die Sie zuvor mit rmic erzeugt haben) schon zu dem Zeitpunkt beim Client vorliegt, an dem er nachschlägt? Sonst wird nämlich der Stub auf dem Client nicht deserialisiert, und die ganze Sache platzt! In einem einfachen System können Sie dem Client die Stub-Klasse einfach »von Hand rüberschieben«.

Aber es gibt noch einen viel cooleren Weg, den zu erklären allerdings den Rahmen dieses Buchs sprengen würde. Falls es Sie interessiert, dieser Weg heißt »dynamischer Klassendownload«. Dabei wird ein Stub-Objekt (d.h. eigentlich *jedes* serialisierte Objekt) mit einer URL »gestempelt«, die dem RMI-System auf dem Client verrät, wo die Klassendatei für dieses Objekt zu finden ist. Falls dann RMI beim Deserialisieren eines Objekts lokal keine Klasse dafür finden kann, holt es sich über diese URL die Klassendatei per HTTP-Get-Anfrage. Sie bräuchten also lediglich einen einfachen Webserver, der die Klassendateien bereithält; außerdem müssten Sie einige Sicherheitsparameter auf dem Client ändern. Es gibt zwar noch einige weitere Probleme bei diesem Verfahren, aber im Wesentlichen funktioniert es so.

## Der vollständige Client-Code

```
import java.rmi.*;

public class MeinRemoteClient {
  public static void main (String[] args) {
    new MeinRemoteClient().los();
  }

  public void los() {

    try {
      MeinRemote service = (MeinRemote) Naming.lookup("rmi://127.0.0.1/RemoteHallo");

      String s = service.sagHallo();

      System.out.println(s);
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Die Klasse Naming (für das Nachschlagen in der RMI-Registrierung) gehört zum Paket java.rmi.

Wird von der Registrierung mit dem Typ Object geliefert – also nicht die Typumwandlung vergessen!

Sie brauchen die IP-Adresse oder den Hostnamen ...

... und den Namen, unter dem der Service >>gebunden<<, d.h. angemeldet wurde.

Sieht genau so aus wie ein stinknormaler Methodenaufruf! (Abgesehen davon, dass die RemoteException berücksichtigt werden muss.)

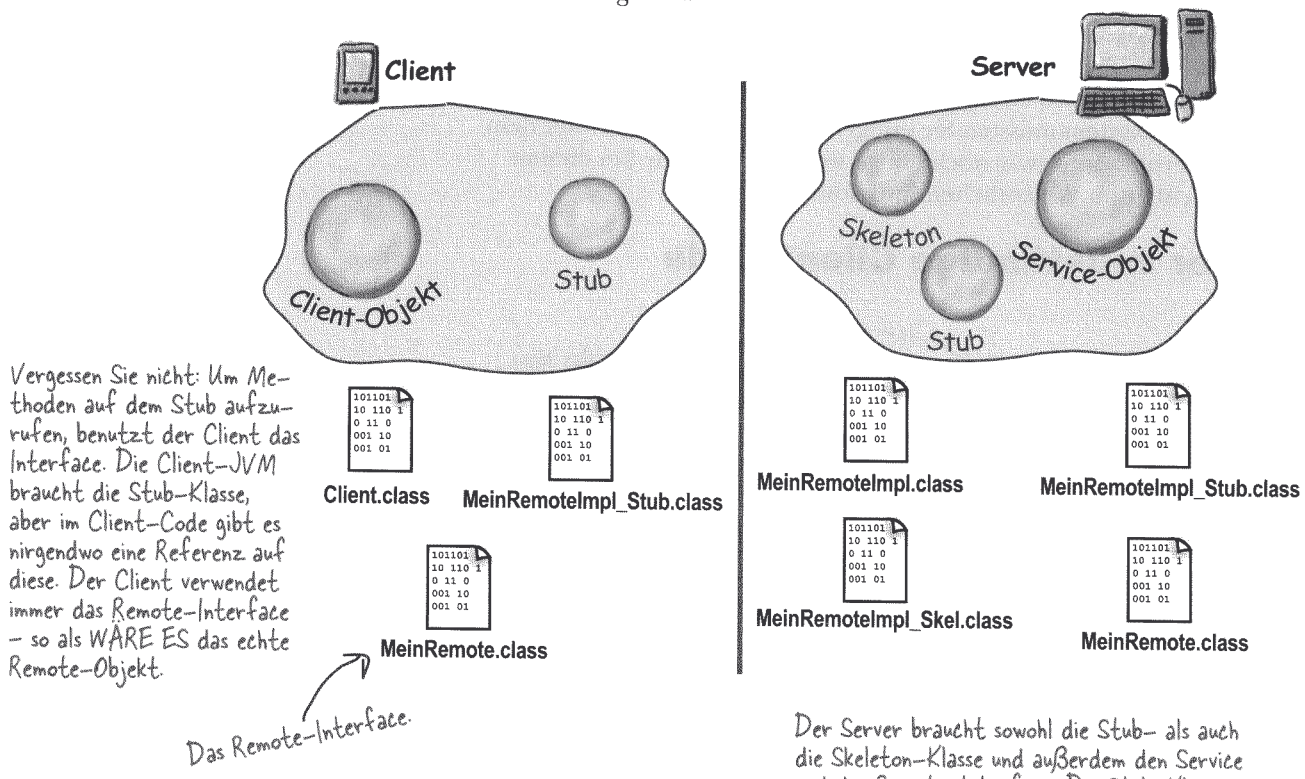
# Vergewissern Sie sich, dass jede Maschine die class-Dateien hat, die sie braucht.

Hier die drei häufigsten Fehler, die Programmierer im Zusammenhang mit RMI machen:

1) Sie vergessen, vor dem Start des Remote-Service rmiregistry zu starten (wenn Sie den Service mit Naming.rebind() registrieren, muss rmiregistry bereits laufen!).

2) Sie vergessen, Argumente und Rückgabetypen serialisierbar zu machen (und das merkt man erst zur Laufzeit, weil der Compiler so etwas nicht bemerkt).

3) Sie vergessen, dem Client die Stub-Klasse zu übergeben.




Vergessen Sie nicht: Um Methoden auf dem Stub aufzurufen, benutzt der Client das Interface. Die Client-JVM braucht die Stub-Klasse, aber im Client-Code gibt es nirgendwo eine Referenz auf diese. Der Client verwendet immer das Remote-Interface – so als WÄRE ES das echte Remote-Objekt.

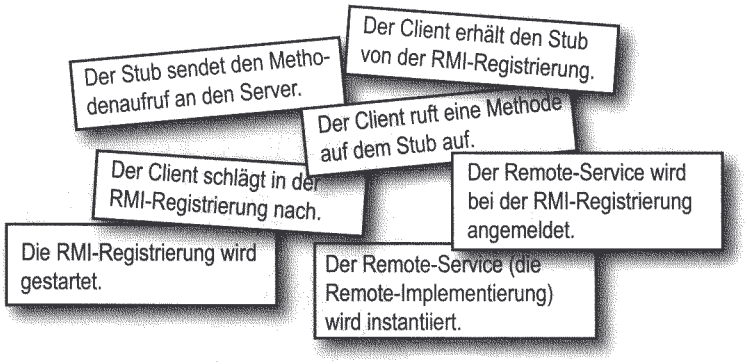
Das Remote-Interface.

Der Server braucht sowohl die Stub- als auch die Skeleton-Klasse und außerdem den Service und das Remote-Interface. Die Stub-Klasse braucht er, weil der wirkliche Service bei seiner Registrierung, wie wir wissen, durch den Stub ersetzt wird.

Spitzen Sie Ihren Bleistift

## Was kam zuerst?

Sehen Sie sich die folgenden Ereignisse an und bringen Sie sie in die Reihenfolge, in der sie in einer Java-RMI-Anwendung eintreten.

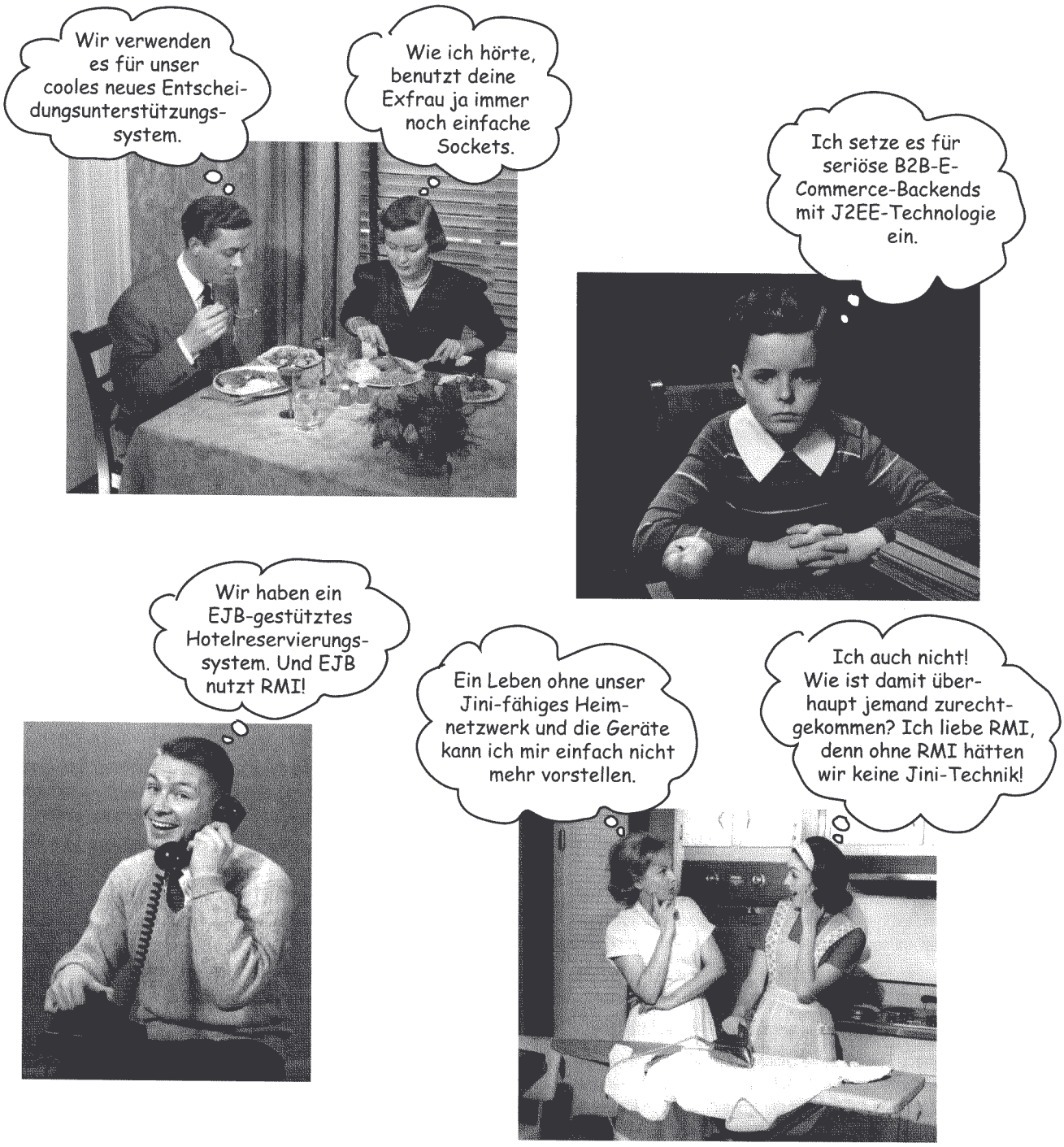


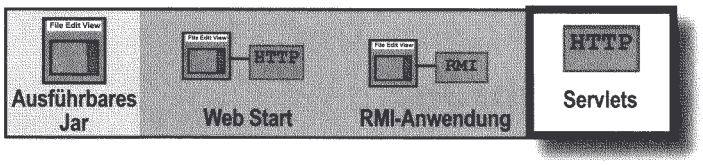

1. 
2. 
3. 
4. 
5. 
6. 
7. 

## Punkt für Punkt

- Ein Objekt auf einem Heap kann keine normale Java-Referenz auf ein Objekt in einem anderen Heap (d.h. ein Objekt, das auf einer anderen JVM läuft) erhalten.
- Mit Java Remote Method Invocation (RMI) können Sie *scheinbar* eine Methode auf einem entfernten Objekt (d.h. einem Objekt in einer anderen JVM) aufrufen – aber es scheint eben nur so.
- Wenn ein Client eine Methode auf einem entfernten Objekt aufruft, ruft er in Wirklichkeit eine Methode auf einem *Proxy* des Remote-Objekts auf. Den Proxy bezeichnet man als »Stub«.
- Ein Stub ist ein Client-Helfer-Objekt, das sich um die Lowlevel-Netzwerk-Einzelheiten (Sockets, Ströme, Serialisierung usw.) kümmert, indem es Methodenaufrufe packt und an den Server sendet.
- Um einen Remote-Service (mit anderen Worten ein Objekt, auf dem ein Remote-Client schließlich Methoden aufrufen kann) zu erstellen, müssen Sie zunächst ein Remote-Interface schreiben.
- Ein Remote-Interface muss das Interface java.rmi.Remote erweitern, und alle Methoden müssen RemoteException deklarieren.
- Ihr Remote-Service implementiert Ihr Remote-Interface.
- Ihr Remote-Service sollte UnicastRemoteObject erweitern. (Ein Remote-Objekt lässt sich technisch auch anders realisieren, aber am einfachsten geht es mit der Erweiterung von UnicastRemoteObject.)
- Ihre Remote-Service-Klasse muss einen Konstruktor haben, der eine RemoteException deklarieren muss (weil dies auch der Konstruktor der Superklasse tut).
- Ihr Remote-Service muss instantiiert und das Objekt bei der RMI-Registrierung angemeldet werden.
- Registrieren Sie einen Remote-Service mit der statischen Methode: `Naming.rebind("Service-Name", serviceInstanz);`
- Die RMI-Registrierung muss auf der gleichen Maschine laufen wie der Remote-Service, und zwar bereits bevor Sie versuchen, ein Remote-Objekt bei der RMI-Registrierung anzumelden.
- Der Client findet Ihren Remote-Service mit Hilfe der statischen Methode: `Naming.lookup("rmi://MeinHostName/Service-Name");`
- Fast alles, was mit RMI zusammenhängt, kann eine (compilergeprüfte) RemoteException auslösen. Dazu gehören auch die Anmeldung oder das Nachschlagen eines Service in der Registrierung und *alle* Remote-Methoden-Aufrufe vom Client auf dem Stub.

# Ja, aber wer *verwendet* denn überhaupt RMI?





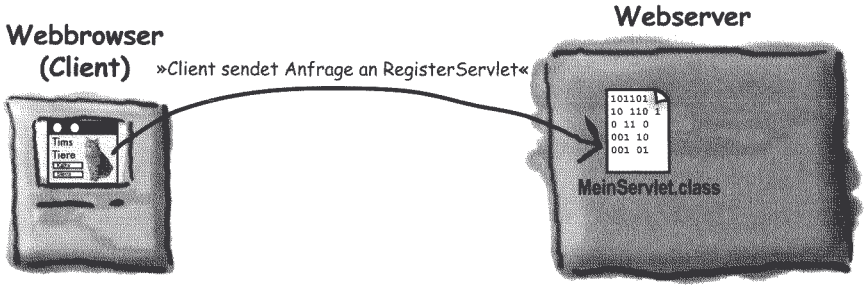

# Was ist mit Servlets?

Servlets sind Java-Programme, die auf (und mit) einem HTTP-Webserver laufen. Wenn ein Client über einen Webbrowser mit einer Webseite kommuniziert, wird eine Anfrage an den Webserver zurückgeschickt. Benötigt die Anfrage die Hilfe eines Java-Servlets, führt der Webserver den Servlet-Code aus (oder ruft ihn auf, falls das Servlet schon läuft). Servlet-Code ist schlicht und einfach Code, der auf dem Server läuft und die Arbeit erledigt, die sich aus der Anfrage des Clients gerade ergibt (z.B. die Speicherung von Informationen in einer Textdatei oder Datenbank auf dem Server). Wenn Sie mit in Perl geschriebenen CGI-Skripten vertraut sind, wissen Sie genau, wovon wir reden. Webentwickler benutzen CGI-Skripten oder Servlets für einfach alles, vom Übermitteln einer vom Benutzer stammenden Info an eine Datenbank bis hin zum Betrieb eines Website-Diskussionsforums.

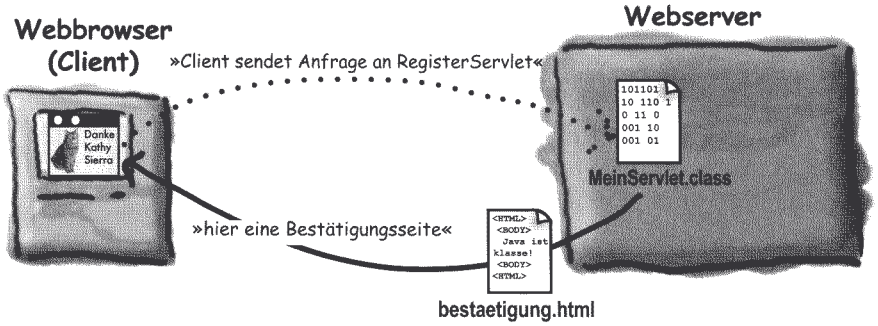
*Und selbst Servlets können RMI verwenden!*

Die bei Weitem häufigste Anwendung der J2EE-Technik ist die Kombination aus Servlets und Enterprise Java Beans (EJBs) – mit den Servlets als Client der EJBs. Und in diesem Fall *setzt das Servlet RMI ein, um mit den EJBs zu kommunizieren.* (Allerdings wird RMI bei EJBs in etwas anderer Weise verwendet, als wir es eben gesehen haben.)

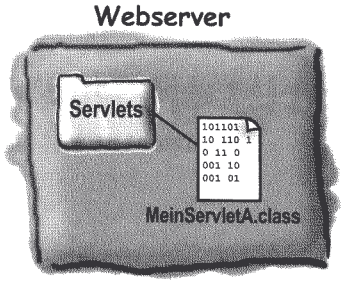
1. Der Client füllt ein Anmeldeformular aus und klickt auf »Absenden«. Der HTTP-Server (d.h. der Webserver) erhält die Anfrage, sieht, dass sie für ein Servlet bestimmt ist, und schickt sie an das Servlet.




2. Das Servlet (Java-Code) wird ausgeführt, schreibt Daten in die Datenbank, baut eine Webseite auf (mit personalisierten Informationen) und sendet diese an den Client zurück, wo sie im Browser angezeigt wird.



# Schritte zur Erstellung und Ausführung eines Servlets




### 1 Herausfinden, wo Sie Ihre Servlets ablegen müssen

Für diese Beispiele wollen wir annehmen, dass Sie bereits einen laufenden Webserver haben und dass er schon für die Unterstützung von Servlets konfiguriert ist. Am wichtigsten ist es herauszufinden, wo genau Ihre Servlet-class-Dateien abgelegt werden müssen, damit Ihr Server sie auch »sieht«. Wenn Ihre Website von einem Internet Service Provider gehostet wird, kann Ihnen die Service-Abteilung sagen, wo Sie Ihre Servlets ablegen sollen (so wie man Ihnen dort auch sagt, wo Ihre CGI-Skripten hingehören).

### 2 Die Datei servlets.jar besorgen und zu Ihrem Klassenpfad hinzufügen

Servlets sind nicht Bestandteil der Java-Standardbibliotheken; Sie brauchen die Klassen, die in servlets.jar gepackt sind. Sie können die Servlet-Klassen von java.sun.com herunterladen oder sie von einem Java-fähigen Webserver erhalten (z.B. Apache Tomcat auf der Site apache.org). Ohne diese Klassen werden Sie Ihre Servlets nicht kompilieren können.

### 3 Eine Servlet-Klasse erstellen (durch Erweiterung von HttpServlet)

Ein Servlet ist einfach eine Java-Klasse, die HttpServlet (aus dem Paket javax.servlet.http) erweitert. Es gibt auch noch andere Arten von Servlets, aber meistens werden für uns nur HttpServlets von Interesse sein.

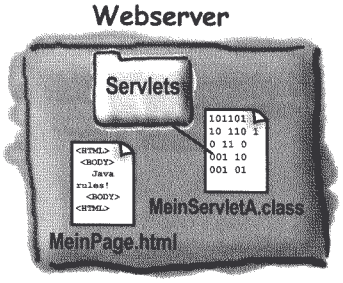
```
public class MeinServletA extends HttpServlet { ... }
```

### 4 Eine HTML-Seite schreiben, die Ihr Servlet aufruft

Wenn der Benutzer auf einen Link klickt, der auf Ihr Servlet verweist, sucht der Webserver das Servlet und ruft dann die geeignete Methode auf, abhängig vom HTTP-Befehl (GET, POST usw.).

```
<a href ="servlets/MeinServletA"> Dies ist ein tolles Servlet.</a>
```

### 5 Ihr Servlet und Ihre HTML-Seite für Ihren Server verfügbar machen




Dies hängt ganz von Ihrem Webserver ab (und genau genommen auch davon, welche *Version* von Java-Servlets Sie verwenden). Ihr Provider sagt Ihnen vielleicht einfach, Sie sollen sie in ein »Servlets«-Verzeichnis auf Ihrer Website legen. Aber wenn Sie beispielsweise die neueste Version von Tomcat verwenden, ist es wesentlich aufwändiger, das Servlet (und die Webseite) an den richtigen Platz zu legen. (Ganz zufällig haben wir auch zu diesem Thema ein Buch ...)

# Ein ganz einfaches Servlet

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MeinServletA extends HttpServlet {

  public void doGet (HttpServletRequest request, HttpServletResponse response)
                                    throws ServletException, IOException {

    response.setContentType("text/html");

    PrintWriter out = response.getWriter();

    String meldung = "Wenn Sie dies lesen, hat es funktioniert!";

    out.println("<HTML><BODY>");
    out.println("<H1>" + meldung + "</H1>");
    out.println("</BODY></HTML>");
    out.close();
  }
}
```

Neben io müssen wir auch zwei der Servlet-Pakete importieren. Denken Sie daran, diese beiden Pakete gehören NICHT zu den Java-Standardbibliotheken – Sie müssen sie extra herunterladen.

Die meisten >>normalen<< Servlets erweitern HttpServlet und überschreiben dann eine oder mehrere Methoden.

Überschreiben Sie bei einfachen HTTP-GET-Meldungen doGet.

Der Webserver ruft diese Methode auf und übergibt die Anfrage des Clients (den >>Request<<, aus dem Sie Daten herausholen können) und ein >>Antwort<<objekt (>>Response<<), mit dessen Hilfe Sie eine Antwort (d.h. eine Seite) zurückschicken können.

Dies teilt dem Server (und dem Browser) mit, von welcher Art das >>Ding<< ist, das der Server als Ergebnis der Servlet-Ausführung zurückschickt.

Vom Response-Objekt bekommen wir einen Ausgabestrom, mit dem wir Informationen zurück an den Server >>schreiben<<.

Was wir >>schreiben<<, ist eine HTML-Seite! Die Seite wird über den Server zurück an den Browser geliefert, genau wie jede andere HTML-Seite, obwohl sie zuvor noch gar nicht existiert hat. Anders ausgedrückt: Es gibt nirgends eine html-Datei mit genau diesem Inhalt.

# HTML-Seite mit einem Link auf dieses Servlet

```
<HTML>
  <BODY>
    <a href="servlets/MeinServletA">Dies ist ein tolles Servlet.</a>
  </BODY>
</HTML>
```

Klicken Sie auf den Link, um das Servlet aufzurufen.

So sieht die Webseite aus:

## Punkt für Punkt

- Servlets sind Java-Klassen, die vollständig auf einem HTTP-(Web-)Server ausgeführt werden.
- Servlets sind von Nutzen, wenn man Code auf dem Server als Resultat aus der Interaktion eines Clients mit einer Webseite ausführen möchte. Wenn beispielsweise ein Client Informationen über ein Webseitenformular sendet, kann das Servlet die Informationen verarbeiten, sie in eine Datenbank eintragen und eine personalisierte Bestätigungsseite zurücksenden.
- Zum Kompilieren eines Servlets brauchen Sie die Servlet-Pakete in der Datei servlets.jar. Die Servlet-Klassen sind nicht Bestandteil der Java-Standardbibliotheken, daher müssen Sie die Datei servlets.jar von jav.sun.com herunterladen oder die Klassen von einem Servlet-fähigen Webserver bekommen. (Anmerkung: Die Servlet-Bibliothek ist in der Java 2 Enterprise Edition (J2EE) enthalten.)
- Um ein Servlet auszuführen, brauchen Sie einen dazu fähigen Webserver, z.B. den Tomcat-Server von apache.org.
- Ihr Servlet muss an einem Ort abgelegt werden, der spezifisch für *Ihren* Webserver ist. Wo das ist, müssen Sie herausfinden, bevor Sie versuchen, Ihre Servlets auszuführen. Wenn Ihre Website bei einem Provider gehostet wird, der Servlets unterstützt, erfahren Sie von diesem, in welches Verzeichnis Sie Ihre Servlets legen sollen.
- Ein typisches Servlet erweitert HttpServlet und überschreibt eine oder mehrere Servlet-Methoden wie z.B. doGet() oder doPost().
- Der Webserver startet das Servlet und ruft abhängig von der Anfrage (Request) des Clients die geeignete Methode (doGet() usw.) auf.
- Das Servlet kann eine Antwort (Response) zurückschicken, indem es sich vom response-Parameter der doGet()-Methode einen PrintWriter-Ausgabestrom geben lässt.
- Das Servlet »schreibt« eine HTML-Seite (komplett mit Tags).

## Es gibt keine Dummen Fragen

**F: Was ist eine JSP, und wie hängt sie mit Servlets zusammen?**

A: JSP steht für Java Server Pages. Letztendlich wandelt der Webserver eine JSP in ein Servlet um; der Unterschied zwischen einem Servlet und einer JSP besteht jedoch darin, was *Sie* (der Entwickler) tatsächlich selbst schreiben. Bei einem Servlet schreiben Sie eine Java-*Klasse*, in deren Ausgabeanweisungen *HTML* steht (falls Sie eine HTML-Seite an den Client zurücksenden). Bei einer JSP ist es dagegen umgekehrt – Sie schreiben eine *HTML*-Seite, die *Java*-Code enthält!

Dies ermöglicht Ihnen die Erstellung dynamischer Webseiten; Sie schreiben die Seite dabei als normale HTML-Seite, aber Sie betten Java-Code (und andere Tags, die zur Laufzeit Java-Code »anstoßen«) darin ein, der zur Laufzeit abgearbeitet wird. Mit anderen Worten, ein Teil der Seite wird erst zur Laufzeit erstellt, wenn der Java-Code ausgeführt wird.

Der Hauptvorteil von JSPs gegenüber normalen Servlets ist, dass es einfach viel leichter ist, den HTML-Teil eines Servlets als JSP-Seite zu schreiben statt mühsam HTML in die print-Anweisungen der Servlet-Antwort zu setzen. Stellen Sie sich eine halbwegs komplexe HTML-Seite vor – und jetzt stellen Sie sich vor, Sie müssen sie innerhalb von println-Anweisungen formatieren. Oje!

Bei vielen Anwendungen braucht man jedoch keine JSPs, weil das Servlet keine dynamische Antwort senden muss oder weil der HTML-Code so einfach ist, dass er nicht viel Arbeit macht. Und es gibt da draußen auch noch eine Menge Webserver, die Servlets unterstützen, aber keine JSPs; daher werden Sie vielleicht keine Wahl haben.

Ein weiterer Vorteil von JSPs ist, dass Sie die Arbeit aufteilen können; Sie lassen die Java-Entwickler die Servlets schreiben und die Webentwickler die JSPs. Zumindest wird das als Vorteil angepriesen. In Wirklichkeit gibt es aber doch bei allen JSP-Schreibern eine Java-Lernkurve (und eine JSP-Tag-Lernkurve); die Annahme, jeder HTML-Webdesigner könnte mal eben schnell JSPs produzieren, ist unrealistisch. Jedenfalls, wenn er keine Tools benutzt. Aber es gibt gute Neuigkeiten: Es kommen langsam Entwicklungswerkzeuge heraus, mit deren Hilfe Webdesigner JSPs erstellen können, ohne den Code von Grund auf selbst schreiben zu müssen.

**F: Ist das alles, was Sie über Servlets sagen? Nachdem Sie hier *so viel* über RMI erzählt haben?**

A: Ja. RMI ist ein Teil der Sprache Java, und alle RMI-Klassen gehören zu den Standardbibliotheken. Servlets und JSPs sind *nicht* Teil der Sprache Java, sondern werden als *Standarderweiterungen* angesehen. RMI können Sie auf jeder neueren JVM ausführen, Servlets und JSPs erfordern hingegen einen korrekt konfigurierten Webserver mit einem Servlet-»Container«. Womit wir sagen wollen: »Das sprengt den Rahmen dieses Buchs.« Aber in dem schönen Buch *Head First Servlets & JSP* können Sie mehr darüber lesen.

# Lassen wir mal spaßeshalber den Phras-O-Mat als Servlet arbeiten

Nachdem wir Ihnen gerade erzählt haben, mehr würden wir zu Servlets nicht sagen, können wir einfach nicht widerstehen, den Phras-O-Mat aus Kapitel 1 zu servletisieren (ja, *selbst das* können wir verbal ausdrücken!). Auch ein Servlet ist schließlich nichts weiter als Java, und Java-Code kann Java-Code aus anderen Klassen aufrufen. Ein Servlet darf daher eine Methode auf dem Phras-O-Mat aufrufen. Sie müssen nichts weiter tun, als die Phras-O-Mat-Klasse in das gleiche Verzeichnis zu legen wie Ihr Servlet, und schon sind Sie im Geschäft. (Der Phras-O-Mat-Code steht auf der nächsten Seite.)

```
import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class KathyServlet extends HttpServlet {
  public void doGet (HttpServletRequest request, HttpServletResponse response)
                                        throws ServletException, IOException {

    String titel = "Phras-O-Mat hat die folgende Phrase gebildet.";

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<HTML><HEAD><TITLE>");
    out.println("Phras-O-Mat");
    out.println("</TITLE></HEAD><BODY>");
    out.println("<H1>" + titel + "</H1>");
    out.println("<P>" + PhrasOMat.machPhrase() + "</P>");
    out.println("<P><a href=\"KathyServlet\">weitere Phrase bilden</a></P>");
    out.println("</BODY></HTML>");

    out.close();
  }
}
```

Sehen Sie? Ihr Servlet kann Methoden auf einer anderen Klasse aufrufen. In diesem Fall rufen wir die statische Methode machPhrase() auf der Klasse PhrasOMat auf (vgl. nächste Seite).

# Phras-O-Mat-Code, Server-freundlich

Diese Version unterscheidet sich etwas von dem Code in Kapitel 1. Im Original haben wir das Ganze in einer main()-Methode ausgeführt und mussten das Programm jedes Mal neu starten, um einen neuen Satz auf der Kommandozeile zu erzeugen. In dieser Version hier gibt der Code einfach einen String (mit der Phrase) zurück, wenn man die statische Methode machPhrase() aufruft. So können Sie die Methode aus jedem anderen Code heraus aufrufen und bekommen einen String mit der zufällig zusammengesetzten Phrase.

Beachten Sie bitte, dass die langen String[]-Array-Zuweisungen hier Opfer des Satzspiegels geworden sind – geben Sie die Bindestriche am Zeilenende nicht mit ein! Tippen Sie einfach immer weiter und überlassen Sie den Zeilenumbruch Ihrem Code-Editor. Und was immer Sie auch tun, drücken Sie niemals mitten in einem String (d.h. irgendwo zwischen den Anführungszeichen) die Eingabetaste.

```
public class PhrasOMat {
  public static String machPhrase() {

    // drei Wortgruppen erstellen, aus denen ausgewählt wird
    String[] wortListeEins = {"verlässliche", "erfolgsorientierte", "webbasierte", "allum-
fassende", "clevere", "kundenorientierte", "pfadkritische", "dynamische", "konkurrenzfähige",
"verteilte", "zielgerichtete"};

    String[] wortListeZwei = {"gepowerte", "haftende", "Mehrwert-", "zentrierte", "gecluster-
te", "proaktive", "Out-of-the-Box", "positionierte", "vernetzte", "fokussierte", "kraftvolle",
"geordnete", "geteilte", "kooperative", "beschleunigte", "Multi-Tier-", "Enterprise-", "B2B-",
"Frontend-"};

    String[] wortListeDrei = {"Schicht", "Endstufe", "Lösung", "Architektur", "Kernkompetenz",
"Strategie", "Kooperation", "Ausrichtung", "Räumlichkeit", "Vision", "Dimension", "Mission"};

    // ermitteln, wie viele Wörter sich in jeder Liste befinden
    int einsLänge = wortListeEins.length;
    int zweiLänge = wortListeZwei.length;
    int dreiLänge = wortListeDrei.length;

    // drei Zufallszahlen generieren, um Wörter aus den Listen herauszuziehen
    int rand1 = (int) (Math.random() * einsLänge);
    int rand2 = (int) (Math.random() * zweiLänge);
    int rand3 = (int) (Math.random() * dreiLänge);

    // eine Phrase zusammenbauen
    String phrase = wortListeEins[rand1] + " " + wortListeZwei[rand2] + " " +
wortListeDrei[rand3];

    // die Phrase zurückgeben
    return ("Was wir brauchen, ist eine " + phrase);
  }
}
```

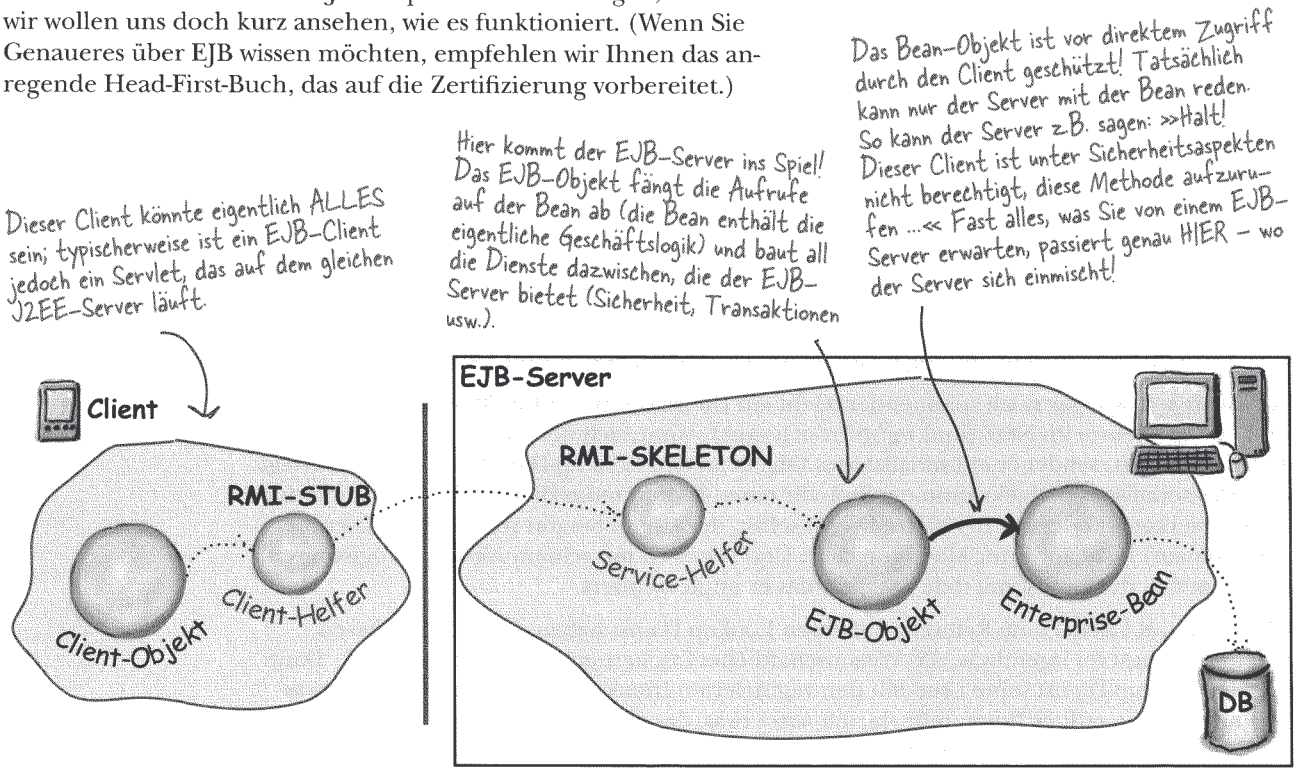
# Enterprise JavaBeans: RMI mit Verstärkung

Für das Schreiben und Ausführen von Remote-Diensten (Remote-Services) ist RMI großartig geeignet. Doch so etwas wie Amazon oder eBay würden Sie nicht ausschließlich auf RMI-Basis betreiben. Bei einer großen, todernsten Unternehmensanwendung brauchen Sie mehr. Sie brauchen etwas, das mit Transaktionen, mit sehr vielen gleichzeitigen Zugriffen (wenn z.B. auf einen Schlag zigtausend Menschen auf Ihren Server zugreifen, um diese Bio-Hundekuchen zu kaufen), mit Sicherheitsproblemen (es soll ja nicht jeder in Ihre Gehaltslistendatenbank hineinkommen können) und mit Datenbankmanagement umgehen kann. Dafür brauchen Sie einen *Unternehmensanwendungsserver* (engl. enterprise application server).

In Java bedeutet das: Sie brauchen einen J2EE-Server (J2EE ist die Abkürzung für Java 2 Enterprise Edition). Dieser umfasst sowohl einen Webserver als auch einen EJB-Server, so dass Sie eine Anwendung ausliefern und einrichten können, die mit Servlets *und* mit EJBs arbeitet. Genau wie Servlets würde auch EJB den Rahmen dieses Buchs sprengen, und es ist nicht möglich, dass wir Ihnen »einfach ein kleines EJB-Beispiel« mit Code zeigen; aber wir wollen uns doch kurz ansehen, wie es funktioniert. (Wenn Sie Genaueres über EJB wissen möchten, empfehlen wir Ihnen das anregende Head-First-Buch, das auf die Zertifizierung vorbereitet.)

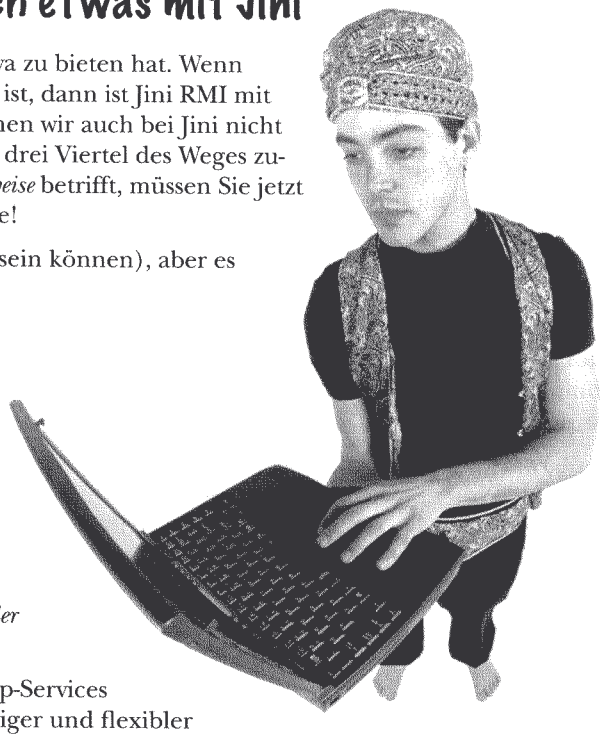
**Ein EJB-Server bietet zusätzlich eine Menge Dienste, die man mit einfachem RMI nicht erhält. Dinge wie Transaktionen, Sicherheit, gleichzeitige Zugriffe, Datenbankmanagement und Netzwerk.**

**Ein EJB-Server springt mitten in einen RMI-Aufruf hinein und bringt dort all seine Dienste mit ins Spiel.**




Dies zeigt nur einen kleinen Ausschnitt aus dem EJB-Geschehen!

# Und zum Abschluss ... zaubern wir noch etwas mit Jini



Wir lieben Jini. Wir halten es für so ungefähr das Beste, was Java zu bieten hat. Wenn EJB RMI mit Verstärkung (durch einen Haufen von Diensten) ist, dann ist Jini RMI mit *Flügeln*. Die reine Java-*Glückseligkeit*. Wie beim Thema EJB können wir auch bei Jini nicht in Details gehen, aber wenn Sie RMI kennen, haben Sie schon drei Viertel des Weges zurückgelegt. Zumindest, was die Technik betrifft. Was die *Denkweise* betrifft, müssen Sie jetzt jedoch einen großen Sprung machen. Nein – *fliegen* müssen Sie!

Jini nutzt RMI (obwohl auch noch andere Protokolle beteiligt sein können), aber es bietet Ihnen zudem ein paar wichtige Eigenschaften wie:

*einen* ***Adaptive Discovery-Mechanismus***

***Selbstheilende Netzwerke***

Bei RMI muss der Client wissen, wie der Remote-Dienst heißt und wo er zu finden ist. Im Client-Code für das Nachschlagen steht sowohl die IP-Adresse oder der Hostname des Remote-Diensts (weil dort die RMI-Registrierung läuft) als auch der logische Name, unter dem der Service registriert wurde.

Aber bei Jini muss der Client nur eins wissen: *welches* ***Interface*** *der Service implementiert!* Mehr nicht.

Wie findet man also etwas? Der Trick dabei sind die Jini-Lookup-Services (d.h. die Nachschlagedienste). Sie sind wesentlich leistungsfähiger und flexibler als die RMI-Registrierung. Zum einen melden sich Jini-Lookup-Dienste *automatisch* selbst beim Netzwerk an. Wenn ein Lookup-Dienst online geht, sendet er eine Nachricht an das Netzwerk (per »IP-Multicast«) und verkündet: »Ich bin hier, falls es jemanden interessiert.«

Aber das ist noch nicht alles. Nehmen wir an, Sie (ein Client) gehen online, *nachdem* sich der Lookup-Dienst angekündigt hat. Dann können *Sie* eine Nachricht an das gesamte Netzwerk schicken und fragen: »Gibt es da draußen irgendwelche Lookup-Dienste?«

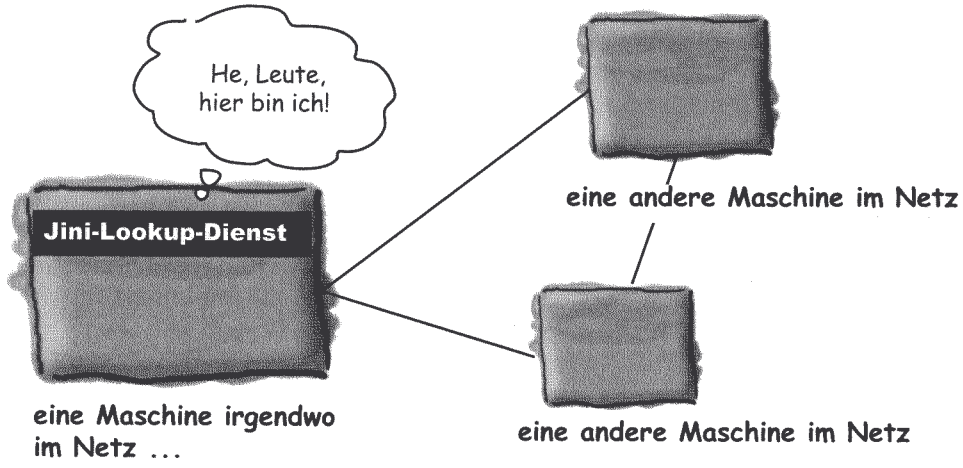
Bloß dass Sie sich für den Lookup-Service *selbst* eigentlich gar nicht interessieren, sondern nur für die Dienste, die beim Lookup-Dienst *registriert* sind. Dinge wie RMI-Remote-Dienste, andere serialisierbare Java-Objekte und sogar Geräte wie Drucker, Kameras und Kaffeemaschinen.

Und hier wird es sogar noch spannender: Geht ein Dienst online, entdeckt er dynamisch alle Jini-Lookup-Dienste im Netz (und *registriert* sich dort). Wenn sich der Dienst beim Lookup-Dienst registriert, sendet er ein serialisiertes Objekt an den Lookup-Dienst, das dort hinterlegt wird. Das serialisierte Objekt kann ein Stub für einen RMI-Remote-Dienst sein, ein Treiber für ein Netzwerkgerät oder gar der ganze Dienst selbst, der (wenn Sie ihn vom Lookup-Dienst erhalten haben) lokal auf Ihrer Maschine läuft. Und statt sich mit seinem *Namen* zu registrieren, registriert sich der Dienst mit dem *Interface,* das er implementiert.

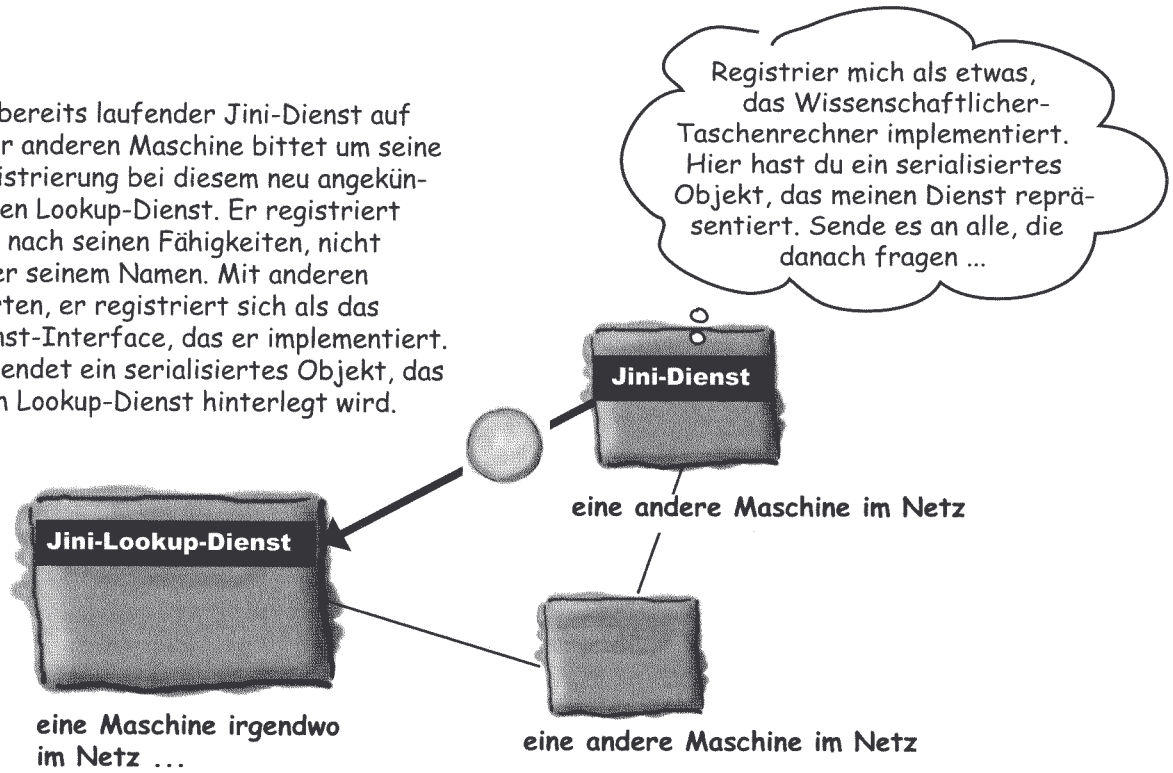
Sobald Sie (als Client) eine Referenz auf einen Lookup-Dienst haben, können Sie diesen fragen: »He, hast du irgendwas, das WissenschaftlicherTaschenrechner implementiert?« Dann geht der Lookup-Dienst seine Liste mit den registrierten Interfaces durch, und falls er etwas Passendes findet, teilt er Ihnen mit: »Ja, ich *habe* etwas, das dieses Interface implementiert. Hier hast du das serialisierte Objekt, das der WissenschaftlicherTaschenrechner-Dienst bei der Registrierung bei mir hinterlegt hat.«

# Das Adaptive Discovery-System in Aktion

1. Der Jini-Lookup-Dienst wird irgendwo im Netzwerk gestartet und kündigt sich selbst per IP-Multicast an.

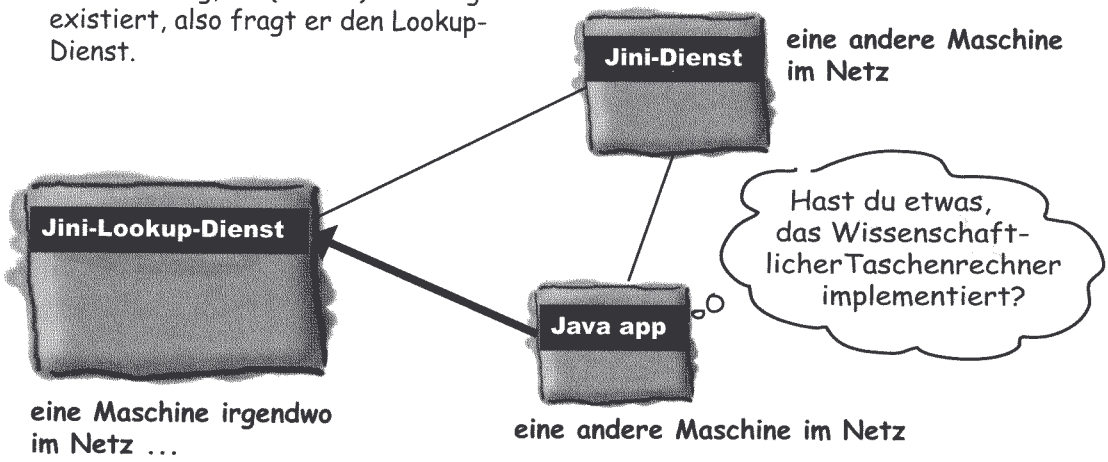



2. Ein bereits laufender Jini-Dienst auf einer anderen Maschine bittet um seine Registrierung bei diesem neu angekündigten Lookup-Dienst. Er registriert sich nach seinen Fähigkeiten, nicht unter seinem Namen. Mit anderen Worten, er registriert sich als das Dienst-Interface, das er implementiert. Er sendet ein serialisiertes Objekt, das beim Lookup-Dienst hinterlegt wird.

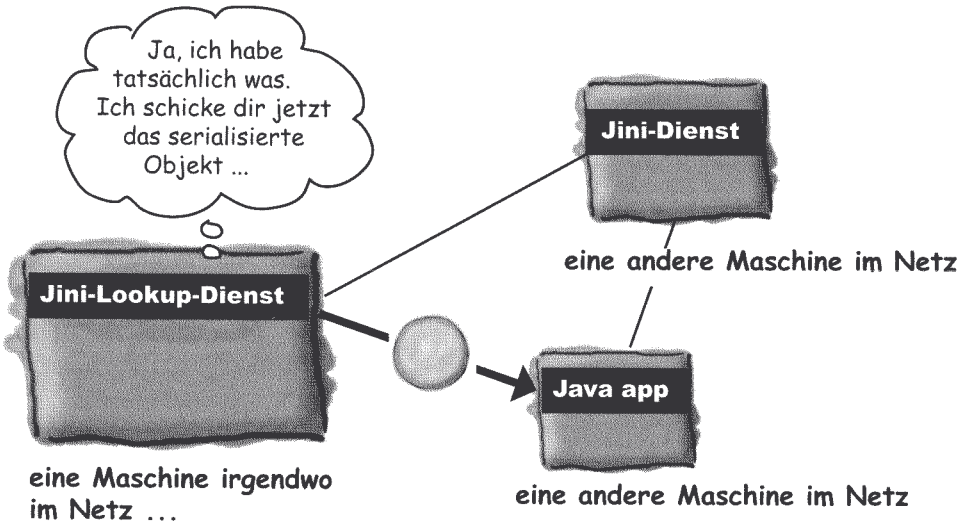


# Das Adaptive Discovery-System in Aktion (Fortsetzung)

③ Ein Client im Netz möchte etwas, das das Interface WissenschaftlicherTaschenrechner implementiert. Er hat keine Ahnung, wo (und ob) das Ding existiert, also fragt er den Lookup-Dienst.

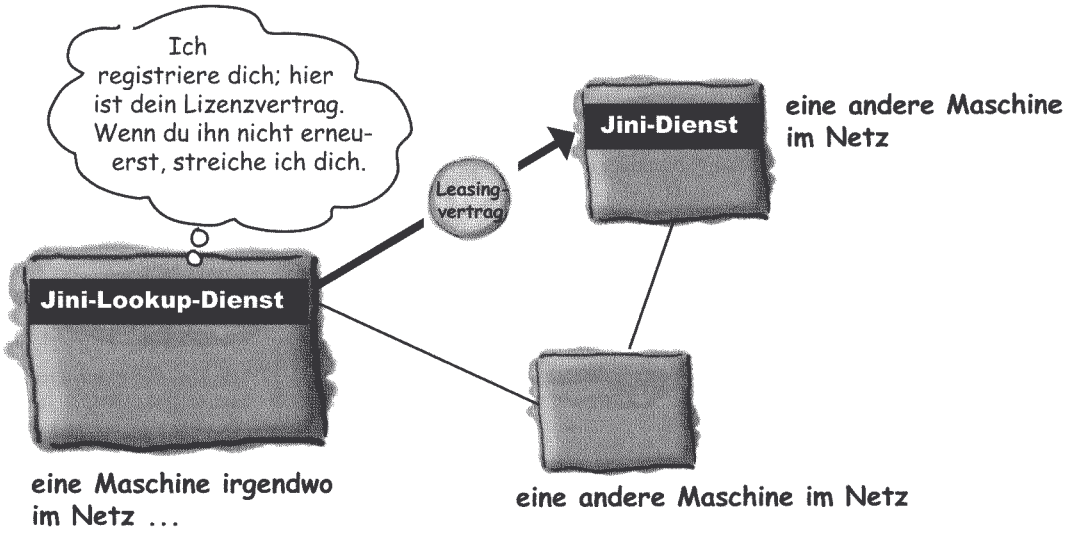



④ Der Lookup-Dienst antwortet, denn er hat tatsächlich etwas, das als Interface WissenschaftlicherTaschenrechner registriert ist.

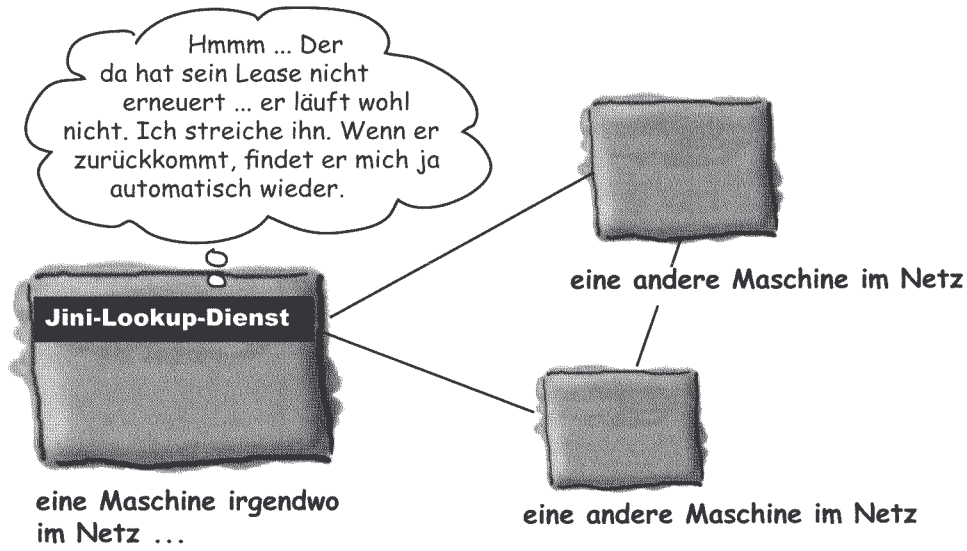


# Selbstheilendes Netzwerk in Aktion

1. Ein Jini-Dienst hat um Registrierung beim Lookup-Dienst gebeten. Dieser schickt ihm als Antwort ein »Lease«. Der neu registrierte Dienst muss das Lease regelmäßig erneuern, sonst nimmt der Lookup-Dienst an, dass der Dienst offline gegangen ist. Der Lookup-Dienst möchte dem Rest des Netzes nämlich immer ein genaues Bild davon geben, welche Dienste verfügbar sind.




2. Der Dienst geht offline (irgendjemand fährt ihn herunter), daher kann er sein Lease beim Lookup-Dienst nicht erneuern. Er wird vom Lookup-Dienst gestrichen.



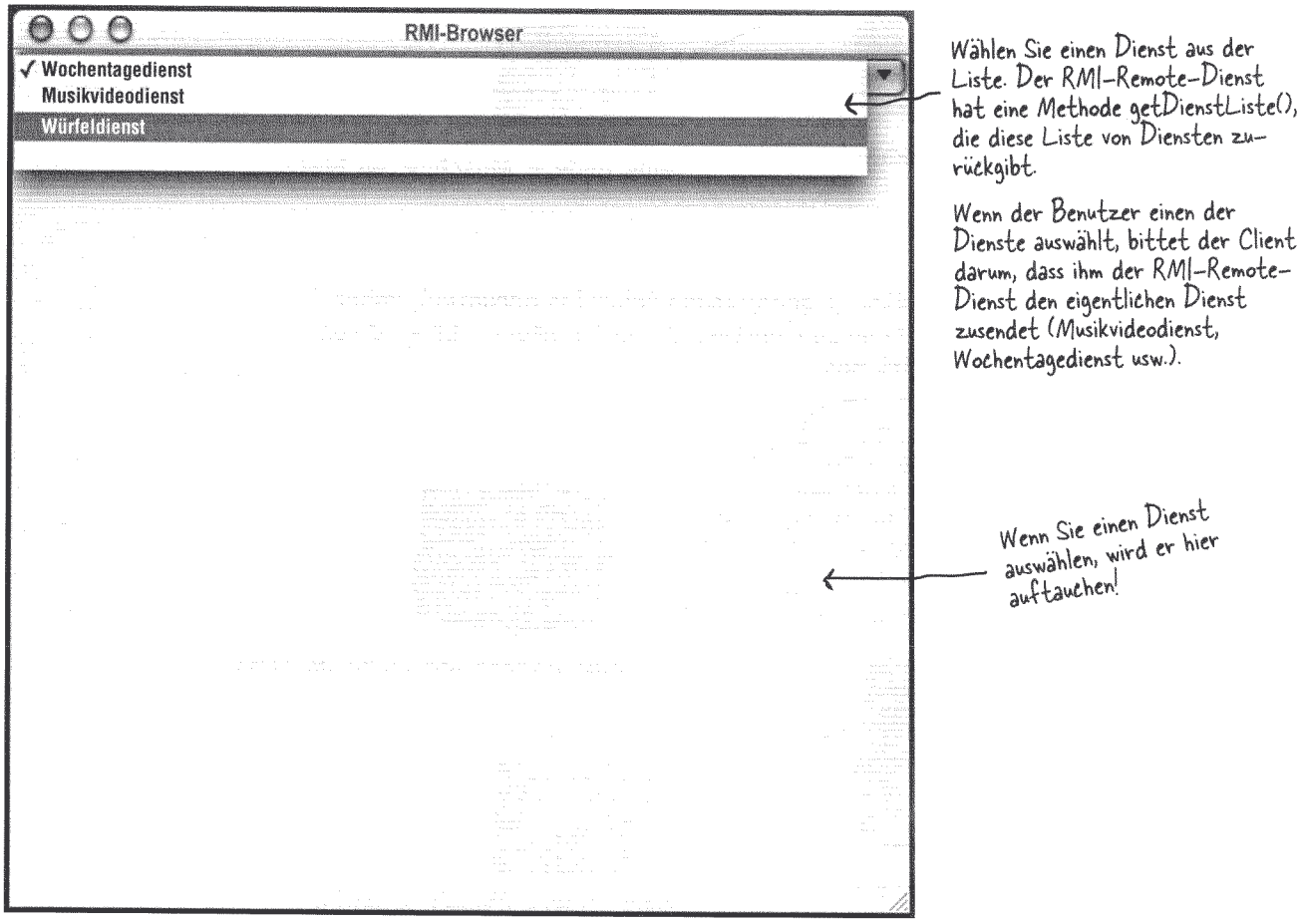
# Abschlussprojekt: Der Universaldienstbrowser

Wir werden etwas programmieren, das zwar nicht Jini-fähig ist, es aber leicht sein könnte. Damit können Sie sich ganz wie bei Jini fühlen, obwohl ausschließlich RMI eingesetzt wird. Der Hauptunterschied zwischen unserer Anwendung und einer Jini-Anwendung besteht eigentlich nur darin, wie der Dienst gefunden wird. Statt des Jini-Lookup-Diensts, der sich automatisch selbst ankündigt und irgendwo im Netz wohnt, verwenden wir die RMI-Registrierung, die auf der gleichen Maschine laufen muss wie der Remote-Dienst und die sich nicht automatisch selbst ankündigt.

Und statt dass sich Ihr Dienst automatisch selbst beim Lookup-Dienst registriert, müssen *wir* ihn bei der RMI-Registrierung anmelden (mit Naming.rebind()).

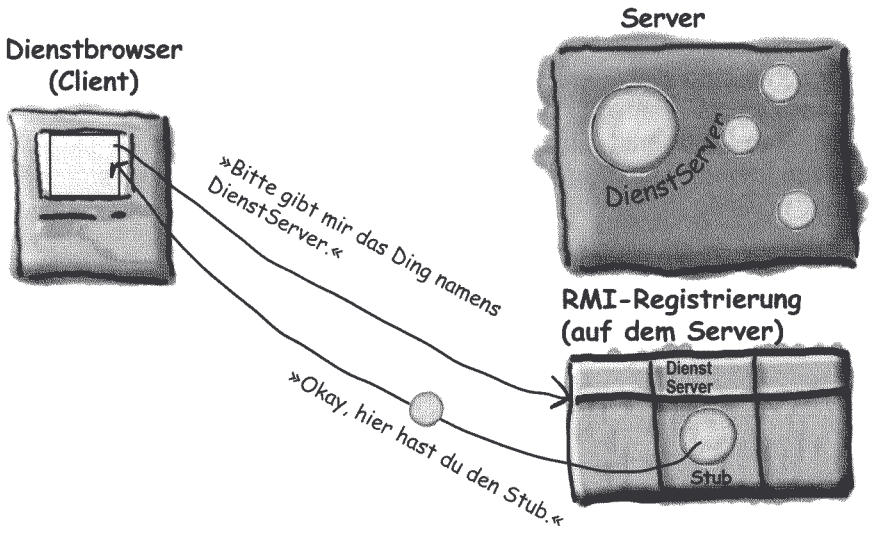
Aber wenn der Client erst einmal den Dienst in der RMI-Registrierung gefunden hat, ist der Rest der Anwendung fast identisch mit dem, was wir in Jini machen würden. (Was fehlt, ist hauptsächlich das »Lease«, mit dem wir ein selbstheilendes Netzwerk hätten, wenn einer der Dienste ausfällt.)

Der Universaldienstbrowser ähnelt einem spezialisierten Webbrowser, aber an Stelle von HTML-Seiten lädt und zeigt uns der Dienstbrowser interaktive Java-Benutzeroberflächen, die wir *Universaldienste* nennen.

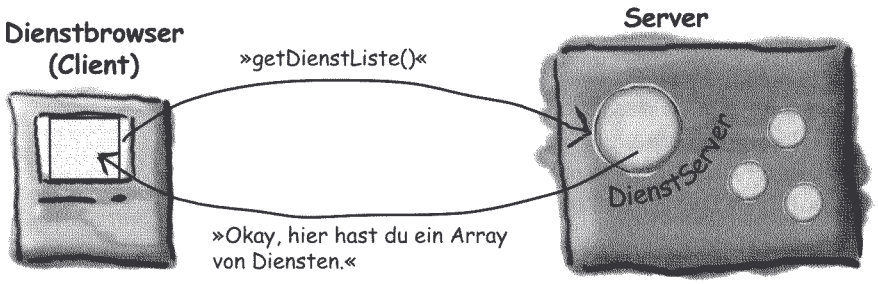


## Wie es funktioniert:

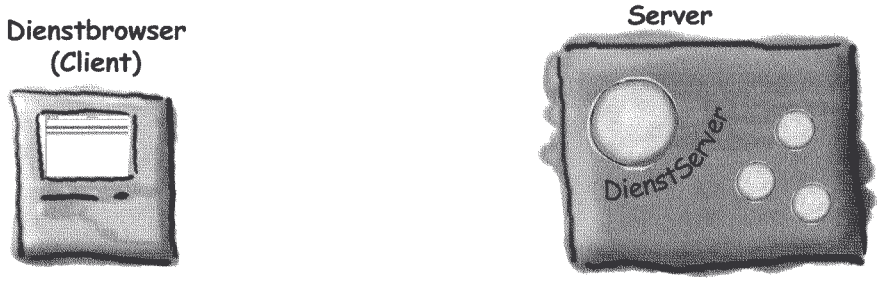
1. Der Client wird gestartet, schlägt in der RMI-Registrierung nach, um den Dienst namens »DienstServer« zu finden, und bekommt den Stub.




2. Der Client ruft getDienstListe() auf. Der DienstServer gibt ein Array von Diensten zurück.

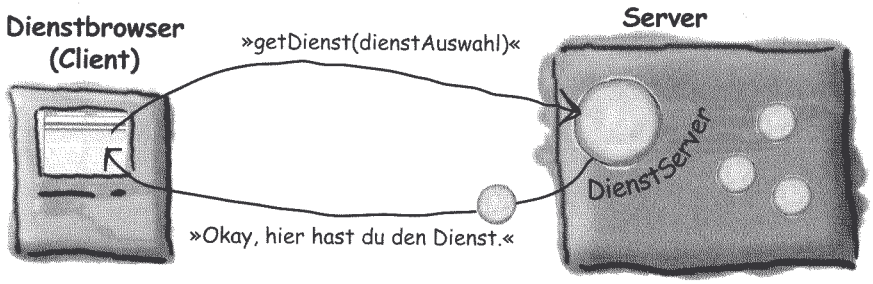



3. Der Client zeigt die Liste der Dienste in einem GUI an.

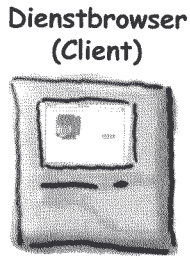


## Wie es funktioniert (Fortsetzung):

④ Der Benutzer wählt etwas aus der Liste aus, also ruft der Client die Methode getDienst() auf dem Remote-Dienst auf. Der Remote-Dienst gibt ein serialisiertes Objekt zurück – einen leibhaftigen Dienst, der innerhalb des Client-Browsers laufen wird.




⑤ Der Client ruft die Methode getGuiPanel() auf dem serialisierten Dienstobjekt auf, das er gerade vom Remote-Dienst erhalten hat. Die GUI für diesen Dienst wird innerhalb des Browsers angezeigt, und der Benutzer kann damit lokal interagieren. Vorläufig brauchen wir daher den Remote-Dienst nicht – so lange, bis sich der Benutzer entschließt, einen anderen Dienst auszuwählen.



# Die Klassen und Interfaces:

1. **interface DienstServer extends Remote**
Ein ganz normales RMI-Remote-Interface für den Remote-Dienst (der Remote-Dienst enthält die Methode, um an die Liste der Dienste zu kommen und einen ausgewählten Dienst zurückzugeben).

2. **class DienstServerImpl implements DienstServer**
Der eigentliche RMI-Remote-Dienst (erweitert UnicastRemoteObject). Seine Aufgabe ist es, all die Dienste (die Dinger, die an den Client versendet werden) zu instantiieren und zu speichern und den Server selbst (DienstServerImpl) bei der RMI-Registrierung anzumelden.

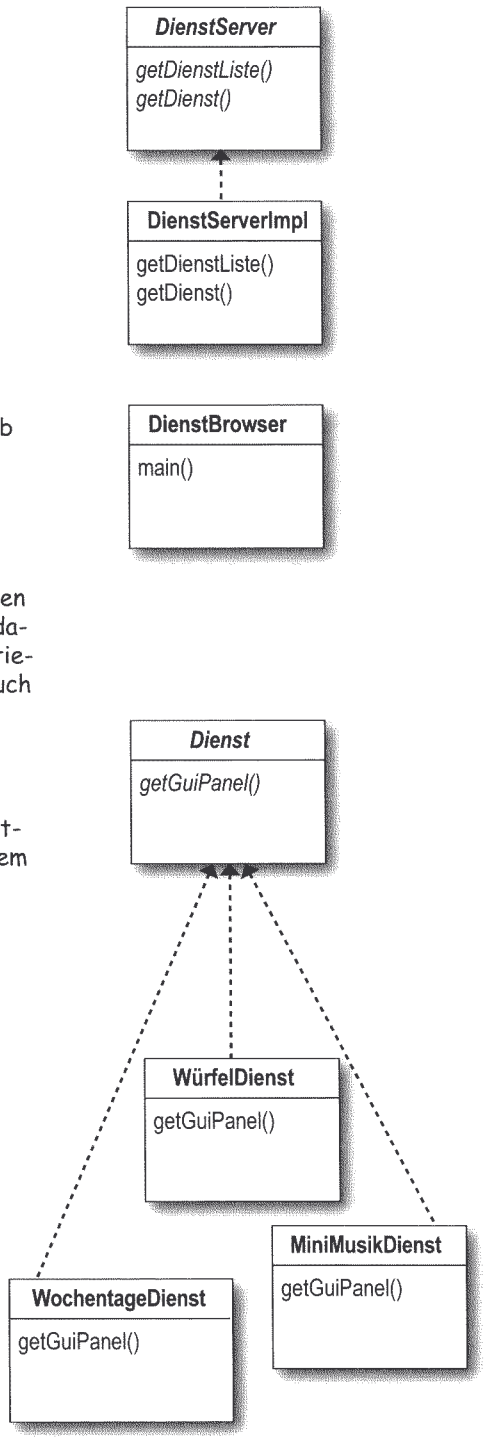
3. **class DienstBrowser**
Der Client. Er erstellt eine sehr einfache Benutzeroberfläche, schlägt in der RMI-Registrierung nach, um an den DienstServer-Stub zu gelangen, und ruft dann eine Remote-Methode darauf auf, um die Liste der Dienste zu erhalten und sie in der GUI-Liste anzuzeigen.

4. **interface Dienst**
Dies ist der Schlüssel zu allem. Dieses sehr einfache Interface enthält nur eine einzige Methode, getGuiPanel(). Jeder Dienst, der an den Client versendet wird, muss dieses Interface implementieren. Erst dadurch wird das Ganze UNIVERSAL einsetzbar! Durch die Implementierung dieses Interface kann ein Service herübergeschickt werden, auch wenn der Client keine Ahnung hat, aus welcher Klasse (oder welchen Klassen) dieser Dienst eigentlich besteht. Der Client weiß lediglich: Was immer auch bei ihm ankommt, es implementiert das Interface Dienst – also MUSS es eine Methode getGuiPanel() haben.
Infolge seines Aufrufs von getDienst(dienstAuswahl) auf dem DienstServer-Stub erhält der Client ein serialisiertes Objekt, und zu diesem Objekt sagt er: »Ich weiß nicht, wer oder was du bist, aber ich WEISS, dass du das Interface Dienst implementierst, also weiß ich auch, dass ich getGuiPanel() auf dir aufrufen kann. Und weil getGuiPanel() ein JPanel zurückgibt, setze ich dieses einfach in die Browseroberfläche und beginne damit zu kommunizieren.«

5. **class WürfelDienst implements Dienst**
Haben Sie Würfel? Falls Sie keine haben und welche brauchen, können Sie diesen Dienst eins bis fünf virtuelle Würfel für Sie werfen lassen.

6. **class MiniMusikDienst implements Dienst**
Erinnern Sie sich noch an dieses sagenhafte kleine »Musikvideo«-Programm aus der ersten GUI-CodeKüche? Wir haben daraus einen Dienst gemacht, und Sie können es immer wieder abspielen – so lange, bis Ihr Besuch endlich geht.

7. **class WochentageDienst implements Dienst**
Sind Sie an einem Freitag geboren? Geben Sie Ihr Geburtsdatum ein und finden Sie es heraus.



## Interface DienstServer (das Remote-Interface)

```
import java.rmi.*;

public interface DienstServer extends Remote {

  Object[] getDienstListe() throws RemoteException;

  Dienst getDienst(Object dienstSchlüssel) throws RemoteException;
}
```

Ein normales RMI-Remote-Interface; es definiert die beiden Methoden, die der Remote-Dienst enthält.

## Interface Dienst (das von den GUI-Diensten implementiert wird)

```
import javax.swing.*;
import java.io.*;

public interface Dienst extends Serializable {
  public JPanel getGuiPanel();
}
```

Ein einfaches altes (Nicht-Remote-)Interface, das die einzige Methode definiert, die jeder Universaldienst haben muss: getGuiPanel(). Das Interface erweitert Serializable, daher ist jede Klasse, die das Interface Dienst implementiert, automatisch serialisierbar.

Das ist ein Muss, weil die Dienste vom Server über das Netz verschickt werden, wenn der Client getDienst() auf dem entfernten Dienst-Server aufgerufen hat.

## Klasse DienstServer (die Remote-Implementierung)

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
```

Eine normale RMI-Implementierung.

```
public class DienstServerImpl extends UnicastRemoteObject implements DienstServer {

  HashMap<String, Dienst> dienstListe;
```

Die Dienste werden in einer HashMap-Collection gespeichert. In diese Collection setzen Sie nicht nur EIN, sondern ZWEI Objekte – ein Schlüssel-Objekt (z.B. einen String) und ein Wert-Objekt (was immer Sie wollen).

```
  public DienstServerImpl() throws RemoteException {
    diensteEinrichten();
  }

  private void diensteEinrichten() {
    dienstListe = new HashMap<String, Dienst>();
    dienstListe.put("Würfeldienst", new WürfelDienst());
    dienstListe.put("Wochentagedienst", new WochentageDienst());
    dienstListe.put("Musikvideodienst", new MiniMusikDienst());
  }
```

Beim Aufruf des Konstruktors werden die eigentlichen Universaldienste initialisiert (WürfelDienst, MiniMusikDienst usw.).

Erstellen Sie die Dienste (die eigentlichen Dienst-Objekte) und setzen Sie sie in die HashMap mit einem String als Namen (für den »Schlüssel«).

```
  public Object[] getDienstListe() {
    System.out.println("in Remote");
    return dienstListe.keySet().toArray();
  }
```

Clients rufen diese Methode auf, um eine Liste von im Browser anzeigbaren Diensten zu erhalten (aus denen der Benutzer einen auswählen kann). Wir übergeben ein Array vom Typ Object (auch wenn es Strings enthält) und erzeugen dafür ein Array, das nur die SCHLÜSSEL aus der HashMap enthält. Ein richtiges Dienst-Objekt schicken wir erst dann, wenn der Client es durch den Aufruf von getDienst() anfordert.

```
  public Dienst getDienst(Object dienstSchlüssel) throws RemoteException {
    Dienst derDienst = dienstListe.get(dienstSchlüssel);
    return derDienst;
  }
```

Der Client ruft diese Methode auf, nachdem der Benutzer einen Dienst aus der angezeigten Liste der Dienste ausgewählt hat (die er wiederum von der obigen Methode erhalten hat). Dieser Code holt sich mit Hilfe des Schlüssels (des gleichen Schlüssels, der ursprünglich an den Client geschickt wurde) den entsprechenden Dienst aus der HashMap.

```
  public static void main (String[] args) {
    try {
      Naming.rebind("DienstServer", new DienstServerImpl());
    } catch(Exception ex) {
      ex.printStackTrace();
    }
    System.out.println("Remote-Dienst läuft");
  }
}
```

## Klasse DienstBrowser (der Client)

```
import java.awt.*;
import javax.swing.*;
import java.rmi.*;
import java.awt.event.*;


public class DienstBrowser {

  JPanel hauptPanel;
  JComboBox dienstListe;
  DienstServer server;


  public void erstelleGUI() {
    JFrame frame = new JFrame("RMI-Browser");
    hauptPanel = new JPanel();
    frame.getContentPane().add(BorderLayout.CENTER, hauptPanel);

    Object[] dienste = getDienstListe();

    dienstListe = new JComboBox(dienste);

    frame.getContentPane().add(BorderLayout.NORTH, dienstListe);

    dienstListe.addActionListener(new MeinListenListener());

    frame.setSize(500,500);
    frame.setVisible(true);

  }

  void ladeDienst(Object dienstAuswahl) {
    try {
      Dienst dienst = server.getDienst(dienstAuswahl);

      hauptPanel.removeAll();
      hauptPanel.add(dienst.getGuiPanel());
      hauptPanel.validate();
      hauptPanel.repaint();
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
```

Diese Methode schlägt in der RMI-Registrierung nach, holt sich den Stub und ruft getDienstListe() auf. (Die Methode selbst steht auf der folgenden Seite.)

Die Dienste (ein Array von Objekten) werden zur JComboBox (der Liste) hinzugefügt. Die JComboBox weiß, wie man aus den Array-Elementen Strings für die Anzeige macht.

Hier fügen wir den eigentlichen Dienst in die Oberfläche ein, nachdem der Benutzer einen Dienst ausgewählt hat. (Diese Methode wird vom EventListener der JComboBox aufgerufen). Wir rufen getDienst() auf dem Remote-Server auf (dem Stub für DienstServer) und übergeben ihm den in der Liste angezeigten String (bei dem es sich um denselben String handelt, den wir ursprünglich beim Aufruf von getDienstListe() vom Server erhalten haben). Der Server gibt den eigentlichen Dienst (serialisiert) zurück, der dann – dank RMI – automatisch deserialisiert wird, und wir rufen einfach getGuiPanel() auf dem Dienst auf und fügen das, was wir erhalten (ein JPanel), zum hauptPanel des Browsers hinzu.

```
  Object[] getDienstListe() {
    Object obj = null;
    Object[] dienste = null;

    try {

      obj = Naming.lookup("rmi://127.0.0.1/DienstServer");

    }
    catch(Exception ex) {
      ex.printStackTrace();
    }
    server = (DienstServer) obj;


    try {

      dienste = server.getDienstListe();

    } catch(Exception ex) {
      ex.printStackTrace();
    }
    return dienste;

  }

  class MeinListenListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {

      Object auswahl =  dienstListe.getSelectedItem();
      ladeDienst(auswahl);
    }
  }

  public static void main(String[] args) {
    new DienstBrowser().erstelleGUI();
  }
}
```

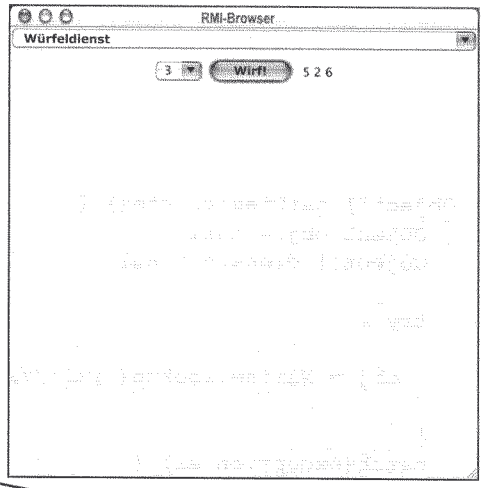
Wir holen uns über das RMI-Lookup den Stub.

Wir wandeln den Typ des Stub in den Remote-Interface-Typ um, um darauf getDienstListe() aufzurufen.

getDienstListe() gibt uns das Array von Objekten, das wir in der JComboBox anzeigen, damit der Benutzer daraus auswählen kann.

Wenn wir hier sind, heißt das, der Benutzer hat etwas in der JComboBox-Liste ausgewählt. Wir nehmen uns also die getroffene Auswahl und laden damit den entsprechenden Dienst (vgl. die Methode ladeDienst() auf der vorherigen Seite, die den Server um den zu dieser Auswahl passenden Dienst bittet).

## Klasse WürfelDienst (ein Universaldienst, implementiert Dienst)

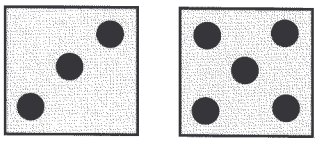



```
import javax.swing.*;
import java.awt.event.*;

public class WürfelDienst implements Dienst {

  JLabel label;
  JComboBox anzahlWürfel;



  public JPanel getGuiPanel() {
    JPanel panel = new JPanel();
    JButton button = new JButton("Wirf!");
    String[] wahl = {"1", "2", "3", "4", "5"};
    anzahlWürfel = new JComboBox(wahl);
    label = new JLabel("hier erscheinen gleich die Augenzahlen");
    button.addActionListener(new WurfListener());
    panel.add(anzahlWürfel);
    panel.add(button);
    panel.add(label);
    return panel;
  }

  public class WurfListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // würfeln
      String augenZahl = "";
      String auswahl = (String)  anzahlWürfel.getSelectedItem();
      int anzahlZuWerfende = Integer.parseInt(auswahl);
      for (int i = 0; i < anzahlZuWerfende; i++) {
        int z = (int) ((Math.random() * 6) + 1);
        augenZahl += (" " + z);
      }
      label.setText(augenZahl);
    }
  }
}
```

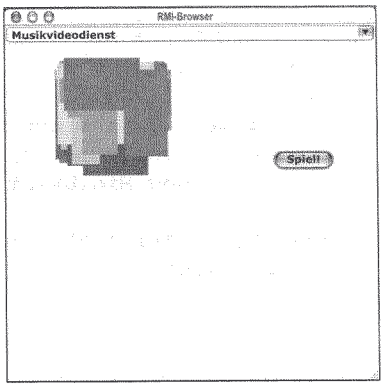
Dies ist DIE wichtige Methode! Die Methode aus dem Interface Dienst, die der Client aufruft, wenn dieser Dienst ausgewählt und geladen wird. In der Methode getGuiPanel() können Sie alles machen, wozu Sie Lust haben, solange Sie ein JPanel zurückgeben, das dann die eigentliche würfelnde GUI erstellt.

### Spitzen Sie Ihren Bleistift

Überlegen Sie sich, wie Sie den WürfelDienst noch verbessern könnten. Ein Vorschlag: Machen Sie die Würfel grafisch, indem Sie das anwenden, was Sie in den GUI-Kapiteln gelernt haben. Benutzen Sie ein Rechteck und zeichnen Sie darauf immer genau die Anzahl von Kreisen, die dem jeweiligen Wurf für den Würfel entspricht.



## Klasse MiniMusikDienst (ein Universaldienst, implementiert Dienst)




```
import javax.sound.midi.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MiniMusikDienst implements Dienst {

  MeinZeichenPanel meinPanel;

  public JPanel getGuiPanel() {
    JPanel hauptPanel = new JPanel();
    meinPanel = new MeinZeichenPanel();
    JButton spielButton = new JButton("Spiel!");
    spielButton.addActionListener(new SpielListener());
    hauptPanel.add(meinPanel);
    hauptPanel.add(spielButton);
    return hauptPanel;
  }

  public class SpielListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {

      try {

        Sequencer sequencer = MidiSystem.getSequencer();
        sequencer.open();

        sequencer.addControllerEventListener(meinPanel, new int[] {127});
        Sequence seq = new Sequence(Sequence.PPQ, 4);
        Track track = seq.createTrack();

        for (int i = 0; i < 100; i+= 4) {

          int zZahl = (int) ((Math.random() * 50) + 1);
          if (zZahl < 38) {  // nur ausführen, wenn zZahl <38 (also in 75 % der Fälle)
            track.add(machEvent(144,1,zZahl,100,i));
            track.add(machEvent(176,1,127,0,i));
            track.add(machEvent(128,1,zZahl,100,i + 2));
          }
        } // Ende der Schleife

        sequencer.setSequence(seq);
        sequencer.start();
        sequencer.setTempoInBPM(220);
      } catch (Exception ex) {ex.printStackTrace();}

    } // actionPerformed schließen
  } // innere Klasse schließen
```

Die Dienst-Methode! Sie tut nichts weiter, als einen Button und den Zeichen-Dienst anzuzeigen (der dann schließlich die Rechtecke zeichnet).

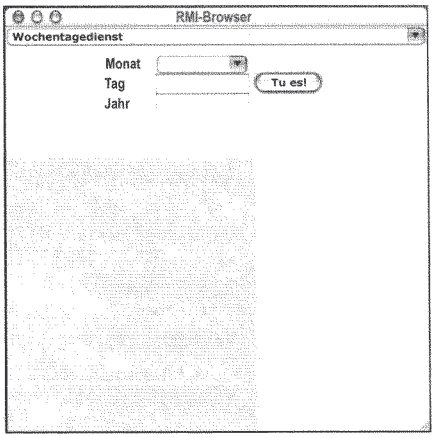
Dies ist all der Musikkram aus der Code-Küche von Kapitel 12, daher schreiben wir die Anmerkungen hier nicht noch mal hin.

## Klasse MiniMusikDienst (Fortsetzung)

```
  public MidiEvent machEvent(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
      ShortMessage a = new ShortMessage();
      a.setMessage(comd, chan, one, two);
      event = new MidiEvent(a, tick);

    }catch(Exception e) { }
    return event;
  }


  class MeinZeichenPanel extends JPanel implements ControllerEventListener {

    // wir zeichnen nur, wenn wir ein Event erhalten haben
    boolean msg = false;

    public void controlChange(ShortMessage event) {
      msg = true;
      repaint();
    }

    public Dimension getPreferredSize() {
      return new Dimension(300,300);
    }

    public void paintComponent(Graphics g) {
      if (msg) {

        Graphics2D g2 = (Graphics2D) g;

        int r = (int) (Math.random() * 250);
        int gr = (int) (Math.random() * 250);
        int b = (int) (Math.random() * 250);

        g.setColor(new Color(r,gr,b));

        int höhe = (int) ((Math.random() * 120) + 10);
        int breite = (int) ((Math.random() * 120) + 10);

        int x = (int) ((Math.random() * 40) + 10);
        int y =  (int) ((Math.random() * 40) + 10);

        g.fillRect(x,y, höhe, breite);
        msg = false;

      } // Ende der if-Verzweigung
    } // Methode schließen
  }  // innere Klasse schließen
} // Klasse schließen
```

Nichts Neues auf dieser Seite. Sie haben das schon alles in der Grafik-CodeKüche gesehen. Wenn Ihnen nach einer weiteren Übung ist, versuchen Sie selbst die Anmerkungen an den Code zu schreiben, und vergleichen Sie diese dann mit der CodeKüche im Kapitel 12.

## Klasse Wochentagedienst (ein Universaldienst, implementiert Dienst)




```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import java.text.*;

public class Wochentagedienst implements Dienst {

  JLabel ausgabeLabel;
  JComboBox monat;
  JTextField tag;
  JTextField jahr;

  public JPanel getGuiPanel() {
    JPanel panel = new JPanel();
    JButton button = new JButton("Tu es!");
    button.addActionListener(new TuEsListener());
    ausgabeLabel = new JLabel("hier erscheint gleich der Wochentag");
    DateFormatSymbols datumsKram = new DateFormatSymbols();
    monat = new JComboBox(datumsKram.getMonths());
    tag = new JTextField(8);
    jahr = new JTextField(8);
    JPanel eingabePanel = new JPanel(new GridLayout(3,2));
    eingabePanel.add(new JLabel("Monat"));
    eingabePanel.add(monat);
    eingabePanel.add(new JLabel("Tag"));
    eingabePanel.add(tag);
    eingabePanel.add(new JLabel("Jahr"));
    eingabePanel.add(jahr);
    panel.add(eingabePanel);
    panel.add(button);
    panel.add(ausgabeLabel);
    return panel;
  }

  public class TuEsListener implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      int monatZahl = monat.getSelectedIndex();
      int tagZahl = Integer.parseInt(tag.getText());
      int jahrZahl = Integer.parseInt(jahr.getText());
      Calendar c = Calendar.getInstance();
      c.set(Calendar.MONTH, monatZahl);
      c.set(Calendar.DAY_OF_MONTH, tagZahl);
      c.set(Calendar.YEAR, jahrZahl);
      Date datum = c.getTime();
      String wochentag = (new SimpleDateFormat("EEEE")).format(datum);
      ausgabeLabel.setText(wochentag);
    }
  }
}
```

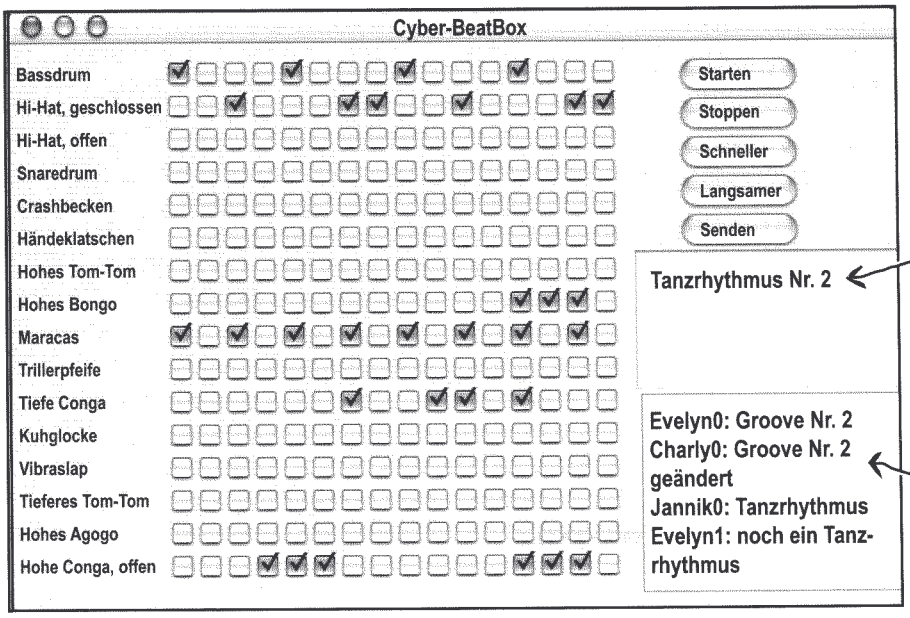
Die Methode aus dem Interface Dienst, die die GUI aufbaut.

Schlagen Sie in Kapitel 10 nach, wenn Sie sich nicht mehr genau daran erinnern, wie Zahlen- und Datumsformatierung funktioniert. Dieser Code ist allerdings etwas anders, weil wir die Klasse Calendar verwenden. Mit Hilfe von SimpleDateFormat können wir außerdem ein Schema für die Ausgabe des Datums angeben.

# Herzlichen Glückwunsch!
## Sie sind am Ende angekommen.

**Da wären aber natürlich noch die beiden Anhänge.**
**Und der Index.**
**Und dann auch noch die Website.**
**Eigentlich gibt es kein Entrinnen ...**

# Anhang A:
# CodeKüchen-Finale




Wenn Sie auf >>Senden<< klicken, wird Ihre Nachricht zusammen mit Ihrem aktuellen Beat-Pattern an die anderen Schlagzeuger geschickt.

Eingehende Nachrichten von Teilnehmern. Klicken Sie auf eine davon, um das Pattern zu laden, das dazugehört, und dann klicken Sie auf >>Start<<, um es abzuspielen.

**Endlich – die vollständige Version der BeatBox!**

**Sie verbindet sich mit einem einfachen MusikServer, damit Sie Beat-Patterns mit anderen Clients austauschen können.**

## Endfassung des BeatBox-Client-Programms

Dieser Code ist größtenteils der gleiche wie der Code aus den CodeKüchen in den vorangegangenen Kapiteln, wir machen daher nicht wieder überall Anmerkungen. Neu ist Folgendes:

GUI: Es werden zwei neue Komponenten hinzugefügt: für den Textbereich (genauer gesagt, die scrollbare Liste), in dem die eingehenden Nachrichten stehen, und für das Textfeld.

NETZWERK: Genau wie der EinfacheChatClient in Kapitel 15 verbindet sich die BeatBox jetzt mit dem Server und bekommt einen Eingabe- und einen Ausgabestrom.

THREADS: Hier starten wir – wieder genau wie bei dem einfachen ChatClient – eine »Reader«-Klasse, die nach eingehenden Nachrichten vom Server Ausschau hält. Die eingehenden Nachrichten enthalten jedoch nicht nur Text, sondern ZWEI Objekte: die String-Nachricht und die serialisierte ArrayList (das Ding, das den Zustand all der Kontrollkästchen enthält).

```
import java.awt.*;
import javax.swing.*;
import java.io.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;
import java.net.*;
import javax.swing.event.*;

public class BeatBoxEndfassung {

  JFrame derFrame;
  JPanel hauptPanel;
  JList eingangsListe;
  JTextField benutzerNachricht;
  ArrayList<JCheckBox> checkboxListe;
  int nummerierung;
  Vector<String> listenVector = new Vector<String>();
  String benutzerName;
  ObjectOutputStream out;
  ObjectInputStream in;
  HashMap<String, boolean[]> andereSeqsMap = new HashMap<String, boolean[]>();

  Sequencer sequencer;
  Sequence sequence;
  Sequence meineSequence = null;
  Track track;

  String[] instrumentNamen = {"Bassdrum", "Hi-Hat, geschlossen", "Hi-Hat, offen",
  "Snaredrum", "Crashbecken", "Händeklatschen", "Hohes Tom-Tom", "Hohes Bongo", "Maracas",
  "Trillerpfeife", "Tiefe Conga", "Kuhglocke", "Vibraslap", "Tieferes Tom-Tom",
  "Hohes Agogo", "Hohe Conga, offen"};

  int[] instrumente = {35,42,46,38,49,39,50,60,70,72,64,56,58,47,67,63};
```

```
public static void main (String[] args) {
  new BeatBoxEndfassung().inBetriebNehmen(args[0]); // args[0] ist Ihr Benutzername
}

public void inBetriebNehmen(String name) {
  benutzerName = name;
  // Verbindung zum Server herstellen
  try {
    Socket sock = new Socket("127.0.0.1", 4242);
    out = new ObjectOutputStream(sock.getOutputStream());
    in = new ObjectInputStream(sock.getInputStream());
    Thread remote = new Thread(new RemoteReader());
    remote.start();
  } catch(Exception ex) {
    System.out.println("Keine Verbindung möglich - Sie müssen allein spielen.");
  }

  midiEinrichten();
  guiErstellen();
}  // inBetriebNehmen schließen

public void guiErstellen() {

  derFrame = new JFrame("Cyber-BeatBox");
  BorderLayout layout = new BorderLayout();
  JPanel hintergrund = new JPanel(layout);
  hintergrund.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

  checkboxListe = new ArrayList<JCheckBox>();

  Box buttonBox = new Box(BoxLayout.Y_AXIS);
  JButton start = new JButton("Starten");
  start.addActionListener(new MeinStartListener());
  buttonBox.add(start);

  JButton stopp = new JButton("Stoppen");
  stopp.addActionListener(new MeinStoppListener());
  buttonBox.add(stopp);

  JButton schneller = new JButton("Schneller");
  schneller.addActionListener(new MeinSchnellerListener());
  buttonBox.add(schneller);

  JButton langsamer = new JButton("Langsamer");
  langsamer.addActionListener(new MeinLangsamerListener());
  buttonBox.add(langsamer);

  JButton senden = new JButton("Senden");
  senden.addActionListener(new MeinSendenListener());
  buttonBox.add(senden);

  benutzerNachricht = new JTextField();
```

Geben Sie auf der Kommandozeile ein Argument für Ihren Chatnamen an.

Beispiel: % java BeatBoxFinal derBlitz

Nichts Neues ... Einrichtung der Netzverbindung, E/A und Erzeugung (und Start) des Reader-Threads.

GUI-Code, nichts Neues.

```
    buttonBox.add(benutzerNachricht);

    eingangsListe = new JList();
    eingangsListe.addListSelectionListener(new MeinListSelectionListener());
    eingangsListe.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    JScrollPane dieListe = new JScrollPane(eingangsListe);
    buttonBox.add(dieListe);
    eingangsListe.setListData(listenVector);

    Box namensBox = new Box(BoxLayout.Y_AXIS);
    for (int i = 0; i < 16; i++) {
      namensBox.add(new Label(instrumentNamen[i]));
    }

    hintergrund.add(BorderLayout.EAST, buttonBox);
    hintergrund.add(BorderLayout.WEST, namensBox);

    derFrame.getContentPane().add(hintergrund);
    GridLayout raster = new GridLayout(16,16);
    raster.setVgap(1);
    raster.setHgap(2);
    hauptPanel = new JPanel(raster);
    hintergrund.add(BorderLayout.CENTER, hauptPanel);

    for (int i = 0; i < 256; i++) {
      JCheckBox c = new JCheckBox();
      c.setSelected(false);
      checkboxListe.add(c);
      hauptPanel.add(c);
    } // Ende der Schleife

    derFrame.setBounds(50,50,300,300);
    derFrame.pack();
    derFrame.setVisible(true);
  } // guiErstellen schließen

  public void midiEinrichten() {
    try {
      sequencer = MidiSystem.getSequencer();
      sequencer.open();
      sequence = new Sequence(Sequence.PPQ,4);
      track = sequence.createTrack();
      sequencer.setTempoInBPM(120);
    } catch(Exception e) {e.printStackTrace();}

  } // midiEinrichten schließen
```

JList ist eine Komponente, die wir bisher noch nicht benutzt haben. Hier werden die eingehenden Nachrichten angezeigt. Anders als in einem normalen Chat, in dem Sie sich die Nachrichten einfach nur ANSEHEN, können Sie in diesem Programm eine Nachricht aus der Liste AUSWÄHLEN und dadurch das angehängte Beat-Pattern laden und abspielen.

Sonst gibt es auf dieser Seite nichts Neues.

Sich den Sequencer beschaffen, eine Sequence und einen Track erzeugen.

```
public void trackErstellenUndStarten() {
  ArrayList<Integer> trackListe = null; // diese Liste enthält die Instrumente
  // für jede vertikale Spalte - mit anderen Worten, für jeden "Tick" (bei dem
  // mehrere Instrumente spielen können)
  sequence.deleteTrack(track);
  track = sequence.createTrack();

  for (int i = 0; i < 16; i++) {

    trackListe = new ArrayList<Integer>();

    for (int j = 0; j < 16; j++) {
      JCheckBox jc = checkboxListe.get(j + (16*i));
      if (jc.isSelected()) {
        int taste = instrumente[i];
        trackListe.add(new Integer(taste));
      } else {
        trackListe.add(null);  // weil dieses Fach im Track leer sein soll
      }
    } // Ende der inneren Schleife
    tracksErzeugen(trackListe);
  } // Ende der äußeren Schleife

  track.add(eventErzeugen(192,9,1,0,16)); // damit wir immer alle 16 Schläge
                                          // komplett durchgehen
  try {
    sequencer.setSequence(sequence);
    sequencer.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);
    sequencer.start();
    sequencer.setTempoInBPM(120);
  } catch(Exception e) {e.printStackTrace();}
} // Methode schließen

public class MeinStartListener implements ActionListener {
  public void actionPerformed(ActionEvent a) {
    trackErstellenUndStarten();
  } // actionPerformed schließen
} // innere Klasse schließen

public class MeinStoppListener implements ActionListener {
  public void actionPerformed(ActionEvent a) {
    sequencer.stop();
  } // actionPerformed schließen
} // innere Klasse schließen

public class MeinSchnellerListener implements ActionListener {
  public void actionPerformed(ActionEvent a) {
    float tempoFactor = sequencer.getTempoFactor();
    sequencer.setTempoFactor((float)(tempoFactor * 1.03));
  } // actionPerformed schließen
} // innere Klasse schließen
```

Wir erstellen einen Track, indem wir die Kontrollkästchen durchgehen, um ihren Zustand abzufragen, und diesen auf ein Instrument abbilden (und das MidiEvent dafür erzeugen). Dies ist ziemlich komplex, aber EXAKT wie in den vorhergehenden Kapiteln. Sehen Sie sich die vollständige Erklärung deshalb in den früheren CodeKüchen an.

Die GUI-Listener. Genau das Gleiche wie in den Versionen aus den früheren Kapiteln.

```
public class MeinLangsamerListener implements ActionListener {
  public void actionPerformed(ActionEvent a) {
    float tempoFactor = sequencer.getTempoFactor();
    sequencer.setTempoFactor((float)(tempoFactor * .97));
  } // actionPerformed schließen
} // innere Klasse schließen

public class MeinSendenListener implements ActionListener {
  public void actionPerformed(ActionEvent a) {
    // eine ArrayList nur für den Zustand der Kontrollkästchen erzeugen
    boolean[] checkboxZustand = new boolean[256];
    for (int i = 0; i < 256; i++) {
      JCheckBox check = (JCheckBox) checkboxListe.get(i);
      if (check.isSelected()) {
        checkboxZustand[i] = true;
      }
    } // Ende der for-Schleife

    try {
      out.writeObject(benutzerName + nummerierung++
              + ": " + benutzerNachricht.getText());
      out.writeObject(checkboxZustand);
    } catch(Exception ex) {
      System.out.println("Tut mir Leid, Senden an Server nicht möglich.");
    }
    benutzerNachricht.setText("");
  } // actionPerformed schließen
} // innere Klasse schließen

public class MeinListSelectionListener implements ListSelectionListener {
  public void valueChanged(ListSelectionEvent le) {
    if (!le.getValueIsAdjusting()) {
      String wahl = (String) eingangsListe.getSelectedValue();
      if (wahl != null) {
        // auf die Map zugreifen und andere Sequence holen
        boolean[] wahlZustand = (boolean[]) andereSeqsMap.get(wahl);
        sequenceWechseln(wahlZustand);
        sequencer.stop();
        trackErstellenUndStarten();
      }
    }
  } // valueChanged schließen
} // innere Klasse schließen
```

Dies ist neu ... größtenteils wie unser EinfacherChatClient, abgesehen davon, dass wir nicht einfach eine String-Nachricht versenden, sondern zwei Objekte serialisieren (die String-Nachricht und das Beat-Pattern) und diese beiden Objekte in den Socket-Ausgabestrom (zum Server) schreiben.

Dies ist ebenfalls neu – ein ListSelectionListener, der uns informiert, wenn der Benutzer etwas aus der Liste der Nachrichten auswählt. Wenn der Benutzer eine Nachricht auswählt, laden wir SOFORT das damit verknüpfte Beat-Pattern (es befindet sich in der als andere-SeqsMap bezeichneten HashMap) und beginnen es abzuspielen. Es gibt ein paar if-Tests wegen kleiner Besonderheiten beim Erhalten von ListSelectionEvents.

Dies ist der Thread-Job – Daten vom Server einlesen. In diesem Code werden die >>Daten<< immer aus zwei serialisierten Objekten bestehen: der String-Nachricht und dem Beat-Pattern (einer ArrayList von Checkbox-Zustandswerten).

```
public class RemoteReader implements Runnable {
  boolean[] checkboxZustand = null;
  String anzuzeigenderName = null;
  Object obj = null;
  public void run() {
    try {
      while((obj=in.readObject()) != null) {
        System.out.println("Object vom Server erhalten");
        System.out.println(obj.getClass());
        String anzuzeigenderName = (String) obj;
        checkboxZustand = (boolean[]) in.readObject();
        andereSeqsMap.put(anzuzeigenderName, checkboxZustand);
        listenVector.add(anzuzeigenderName);
        eingangsListe.setListData(listenVector);
      } // Ende der while-Schleife
    } catch(Exception ex) {ex.printStackTrace();}
  } // run schließen
} // innere Klasse schließen
```

Wenn eine Nachricht hereinkommt, lesen (deserialisieren) wir die beiden Objekte (die Nachricht und die ArrayList mit Booleschen Checkbox-Zustandswerten) und fügen sie zur JList-Komponente hinzu. Das Hinzufügen zu einer JList erfolgt in zwei Schritten: Sie haben einen Vector mit den Daten der Liste (Vector ist eine altmodische Form von ArrayList) und weisen dann die JList an, diesen Vector als Quelle für das zu benutzen, was sie in der Liste anzeigen soll.

Diese Methode wird aufgerufen, wenn der Benutzer etwas aus der Liste auswählt. Das Beat-Pattern wird SOFORT durch das ausgewählte ersetzt.

```
public void sequenceWechseln(boolean[] checkboxZustand) {
  for (int i = 0; i < 256; i++) {
    JCheckBox check = (JCheckBox) checkboxListe.get(i);
    if (checkboxZustand[i]) {
      check.setSelected(true);
    } else {
      check.setSelected(false);
    }
  } // Ende der for-Schleife
} // sequenceWechseln schließen
```

Der ganze MIDI-Kram ist genau der gleiche wie in der vorhergehenden Version.

```
public void tracksErzeugen(ArrayList liste) {
  Iterator it = liste.iterator();
  for (int i = 0; i < 16; i++) {
    Integer num = (Integer) it.next();
    if (num != null) {
      int numTaste = num.intValue();
      track.add(eventErzeugen(144,9,numTaste,100, i));
      track.add(eventErzeugen(128,9,numTaste,100, i + 1));
    }
  } // Ende der for-Schleife
} // tracksErzeugen schließen
```

```
  public MidiEvent eventErzeugen(int comd, int chan, int one, int two, int tick) {
    MidiEvent event = null;
    try {
      ShortMessage a = new ShortMessage();
      a.setMessage(comd, chan, one, two);
      event = new MidiEvent(a, tick);
    }catch(Exception e) { }
    return event;
  } //eventErzeugen schließen

} // Klasse schließen
```

Nichts Neues. Genau wie die letzte Fassung.

## Spitzen Sie Ihren Bleistift

Auf welche Weise könnten Sie dieses Programm noch verbessern?

Hier ein paar Ideen, mit denen Sie anfangen könnten:

1) Wenn Sie ein Pattern auswählen, wird das Pattern, das Sie vorher gespielt haben, vom Tisch gefegt. Falls das ein neues Pattern war, an dem Sie gerade gearbeitet haben (oder eine Modifikation eines anderen), haben Sie Pech gehabt. Vielleicht möchten Sie, dass ein Dialogfeld erscheint, das den Benutzer fragt, ob er das aktuelle Pattern speichern will.

2) Wenn Sie vergessen, ein Kommandozeilen-Argument einzugeben, erhalten Sie beim Ausführen einfach eine Exception! Schreiben Sie etwas in die main-Methode, das überprüft, ob ein Kommandozeilen-Argument übergeben wurde. Wenn der Benutzer keins angibt, wählen Sie ein vordefiniertes oder geben eine Meldung darüber aus, dass der Benutzer es erneut ausführen soll, aber dieses Mal mit einem Argument für seinen Chatnamen.

3) Vielleicht wäre es schön, wenn man die Möglichkeit hätte, durch Klick auf einen Button ein zufallsgeneriertes Pattern zu erhalten. Vielleicht stoßen Sie dabei auf eins, das Ihnen wirklich gefällt. Noch besser wäre es, wenn man zudem schon existierende »Grund«-Patterns laden könnte, z.B. für Jazz, Rock, Reggae usw., die der Benutzer dann erweitern kann.

## Endfassung des BeatBox-Server-Programms

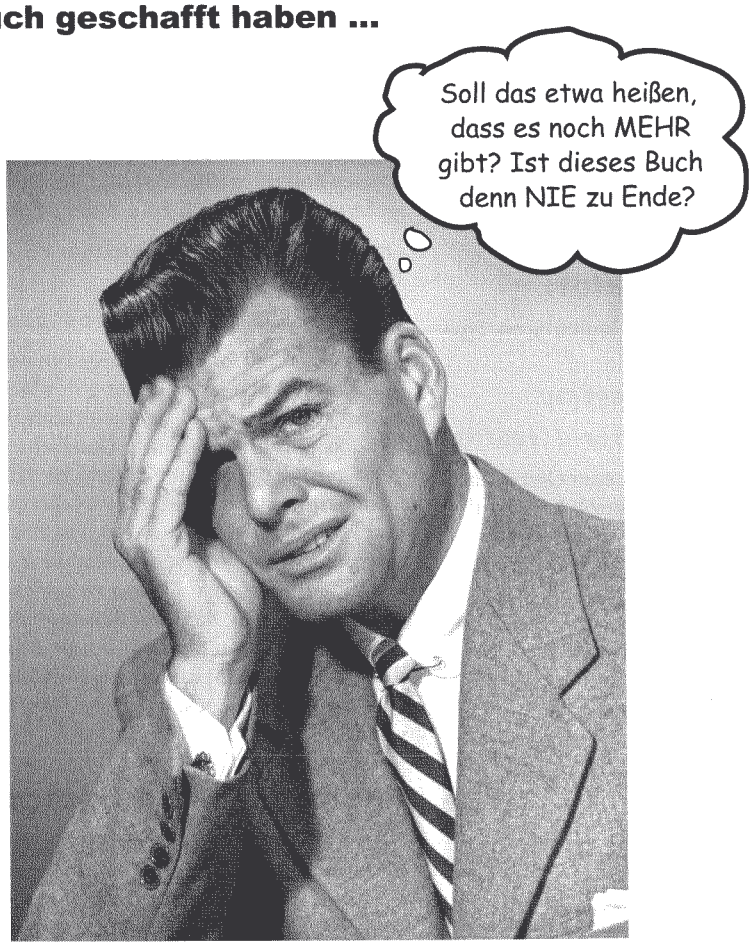
Dieser Code ist größtenteils identisch mit dem EinfacherChatServer-Code, den wir im Kapitel »Netzwerkprogrammierung und Threads« geschrieben haben. Tatsächlich besteht der einzige Unterschied darin, dass der Server *zwei* serialisierte Objekte empfängt und weiterversendet statt nur einen einfachen String (eines der beiden serialisierten Objekte ist allerdings zufällig auch ein String).

```
import java.io.*;
import java.net.*;
import java.util.*;

public class MusikServer {

  ArrayList<ObjectOutputStream> clientOutputStreams;

  public static void main (String[] args) {
    new MusikServer().los();
  }

  public class ClientHandler implements Runnable {

    ObjectInputStream in;
    Socket clientSocket;

    public ClientHandler(Socket socket) {
      try {
        clientSocket = socket;
        in = new ObjectInputStream(clientSocket.getInputStream());

      } catch(Exception ex) {ex.printStackTrace();}
    } // Konstruktor schließen

    public void run() {
      Object o2 = null;
      Object o1 = null;
      try {

        while ((o1 = in.readObject()) != null) {

          o2 = in.readObject();

          System.out.println("zwei Objekte gelesen");
          esAllenMitteilen(o1, o2);
        } // Ende der while-Schleife

      } catch(Exception ex) {ex.printStackTrace();}
    } // run schließen
  } // innere Klasse schließen
```

```
    public void los() {
      clientOutputStreams = new ArrayList<ObjectOutputStream>();

      try {
        ServerSocket serverSock = new ServerSocket(4242);

        while(true) {
          Socket clientSocket = serverSock.accept();
          ObjectOutputStream out = new ObjectOutputStream(clientSocket.getOutputStream());
          clientOutputStreams.add(out);

          Thread t = new Thread(new ClientHandler(clientSocket));
          t.start();

          System.out.println("Verbindung hergestellt");
        }
      }catch(Exception ex) {
        ex.printStackTrace();
      }
    } // los schließen

    public void esAllenMitteilen(Object eins, Object zwei) {
      Iterator it = clientOutputStreams.iterator();
      while(it.hasNext()) {
        try {
          ObjectOutputStream out = (ObjectOutputStream) it.next();
          out.writeObject(eins);
          out.writeObject(zwei);
        }catch(Exception ex) {ex.printStackTrace();}
      }
    } // esAllenMitteilen schließen

} // Klasse schließen
```

# Anhang B

## Die Top Ten der Themen, die es nicht ins Buch geschafft haben ...




Wir haben eine ganze Menge Stoff behandelt, und Sie haben das Buch fast abgeschlossen. Sie werden uns fehlen. Aber noch können wir Sie nicht ganz gehen lassen. Wir hätten ein schlechtes Gefühl, wenn wir Sie so schon nach JavaLand gehen lassen würden. Etwas Vorbereitung brauchen Sie noch. Es ist unmöglich, alles, was Sie wissen müssen, in diesen relativ schmalen Anhang zu pressen. *Ursprünglich* hatten wir versucht, hier alles reinzupacken, was Sie über Java wissen müssen (und wir in den anderen Kapiteln noch nicht behandelt haben). Dazu haben wir die Punktgröße der Schrift auf .00003 verkleinert. Dann hat zwar alles gepasst, aber leider konnte es niemand mehr lesen. Deswegen haben wir das Meiste weggeworfen und die besten Teile für diesen Top Ten-Anhang aufgehoben.

Das *ist* wirklich das Ende des Buchs. Abgesehen vom Index (den Sie natürlich unbedingt lesen sollten!).

# Platz 10: Bit-Manipulationen

## Warum Sie das interessiert?

Wir haben erwähnt, dass ein byte 8 Bit hat, ein short 16 Bit und so weiter. Gelegentlich möchten Sie vielleicht einzelne Bits an- und ausschalten. Vielleicht schreiben Sie ja gerade Code für Ihren neuen Java-fähigen Toaster und stellen fest, dass einige Toaster-Einstellungen auf Grund starker Speicherbeschränkungen nur auf Bit-Ebene gesteuert werden können. (Der Lesbarkeit halber zeigen wir in den Kommentaren nur die letzten 8 Bit und nicht die ganzen 32 Bit eines ints).

## Der bitweise NICHT-Operator: ~

Dieser Operator »schaltet alle Bits« eines elementaren Typs um.

```
int x = 10; // die Bits sind 00001010
x = ~x;     // jetzt sind die Bits 11110101
```

Die nächsten drei Operatoren vergleichen zwei elementare Typen Bit für Bit und liefern ein Ergebnis zurück, das auf dem Vergleich dieser Bits basiert. Das folgende Beispiel werden wir für die nächsten drei Operatoren verwenden:

```
int x = 10; // die Bits sind 00001010
int y = 6;  // die Bits sind 00000110
```

## Der bitweise UND-Operator: &

Dieser Operator liefert einen Wert zurück, dessen Bits nur eingeschaltet sind, wenn das entsprechende Bit in ***beiden*** Operanden eingeschaltet ist:

```
int a = x & y; // die Bits sind 00000010
```

## Der bitweise ODER-Operator: |

Dieser Operator liefert einen Wert zurück, dessen Bits nur eingeschaltet sind, wenn das entsprechende Bit bei ***einem*** der Operanden eingeschaltet ist:

```
int a = x | y; // die Bits sind 00001110
```

## Der bitweise XODER-Operator (exklusives ODER): ^

Dieser Operator liefert einen Wert zurück, dessen Bits nur eingeschaltet sind, wenn die entsprechenden Bits bei ***nur einem*** der Operanden eingeschaltet sind:

```
int a = x ^ y; // die Bits sind 00001100
```

## Die Verschiebe-Operatoren (Shift-Operatoren)

Diese Operatoren nehmen einen einzigen elementaren Ganzzahltyp und verschieben alle Bits darin in die eine oder die andere Richtung. Wenn Sie Ihre Fertigkeiten in Binärarithmetik etwas entstauben wollen, bemerken Sie vielleicht, dass das Verschieben der Bits nach *links* eine Zahl mit einer Zweier-Potenz *multipliziert* und das Verschieben der Bits nach *rechts* eine Zahl durch eine Zweier-Potenz *dividiert*.

Das folgende Beispiel werden wir für die nächsten drei Operatoren verwenden:

```
int x = -11; // die Bits sind 11110101
```

Okay, okay! Wir haben es lange hinausgezögert, aber jetzt kommt die kürzeste Erklärung der Welt zur Speicherung negativer Zahlen und zu Zweierkomplementen. Erinnern Sie sich: Das linke Bit eines Ganzzahltyps wird als das ***Vorzeichenbit*** bezeichnet. In Java ist bei einer negativen Zahl das Vorzeichenbit *immer eingeschaltet* (d.h. auf 1 gesetzt). Bei einer positiven Ganzzahl und 0 ist das Vorzeichenbit immer *ausgeschaltet* (0). Java verwendet die Zweierkomplement-Formel, um negative Zahlen zu speichern. Wenn Sie das Vorzeichen einer Zahl mit dem Zweierkomplement ändern möchten, schalten Sie einfach alle Bits um und addieren dann 1 hinzu (bei einem byte-Wert hieße das beispielsweise, dass man dem umgeschalteten Wert 00000001 hinzufügen würde).

## Rechts-Schiebe-Operator: >>

Dieser Operator verschiebt alle Bits einer Zahl um eine bestimmte Anzahl von Stellen nach rechts und füllt alle Bits auf der linken Seite mit dem ursprünglichen Wert des Bits ganz links außen auf. **Das Vorzeichenbit ändert sich *nicht*:**

```
int y = x >> 2; // die Bits sind 11111101
```

## Vorzeichenloser Rechts-Schiebe-Operator: >>>

Wie der Rechts-Schiebe-Operator, füllt die Bits ABER IMMER mit Nullen. **Das Vorzeichenbit *könnte* sich ändern**:

```
int y = x >>> 2; // die Bits sind 00111101
```

## Links-Schiebe-Operator: <<

Wie der vorzeichenlose Rechts-Schiebe-Operator, verschiebt aber in die andere Richtung. Die Bits werden von rechts aus mit Nullen aufgefüllt. **Das Vorzeichenbit *könnte* sich ändern.**

```
int y = x << 2; // die Bits sind 11010100
```

# Platz 9: Unveränderbarkeit

## Warum es Sie interessiert, dass Strings unveränderbar sind?

Wenn Ihr Java-Programm größer wird, ist es unausweichlich, dass Sie massenhaft String-Objekte haben. Aus Sicherheitsgründen und um Speicherplatz zu sparen (denken Sie daran, dass Ihr Java-Programm auf winzigen Java-fähigen Handys laufen könnte), sind Strings in Java unveränderbar. Das bedeutet, dass folgender Code

```
String s = "0";
for (int x = 1; x < 10; x++) {
s = s + x;
}
```

eigentlich bewirkt, dass Sie 10 String-Objekte (mit den Werten »0«, »01«, »012« bis »0123456789«) erzeugen. Am Ende verweist *s* auf den String mit dem Wert »0123456789«. Aber zu diesem Zeitpunkt gibt es *zehn* Strings!

Jedes Mal, wenn Sie einen neuen String machen, steckt die JVM ihn in einen besonderen Speicherbereich, der als String-Pool bezeichnet wird (klingt erfrischend, oder?). Wenn es im String-Pool bereits einen String mit dem gleichen Wert gibt, erzeugt die JVM keinen zweiten String mit diesem Wert, sondern lässt Ihre Referenzvariable einfach auf den vorhandenen Eintrag verweisen. Darauf kann sich die JVM beschränken, weil Strings unveränderbar sind. Eine Referenzvariable kann nicht im Rücken einer anderen Referenzvariablen, die auf den gleichen String verweist, den Wert eines Strings ändern.

Die andere Sache mit dem String-Pool ist, dass der Garbage Collector *nicht dort hingeht*. In unserem Beispiel würden die ersten neun Strings, die in unserer *for*-Schleife erzeugt wurden, für immer nutzlos herumsitzen und Speicherplatz verschwenden, es sei denn, Sie machen später zufälligerweise einen String mit einem gleichen Wert (»01234« beispielsweise).

### Wie das Speicher spart?

Ähm. Wenn Sie nicht aufpassen, *tut es das nicht*! Aber wenn Sie wissen, wie die Unveränderbarkeit von Strings funktioniert, können Sie das manchmal ausnutzen, um Speicher zu sparen. Müssen Sie viel String-Bearbeitungen durchführen, (Verkettungen usw.), gibt es allerdings eine Klasse, die für diesen Zweck besser geeignet ist. Über StringBuilder werden wir in ein paar Seiten sprechen.

## Warum es Sie interessiert, dass Wrapper unveränderbar sind?

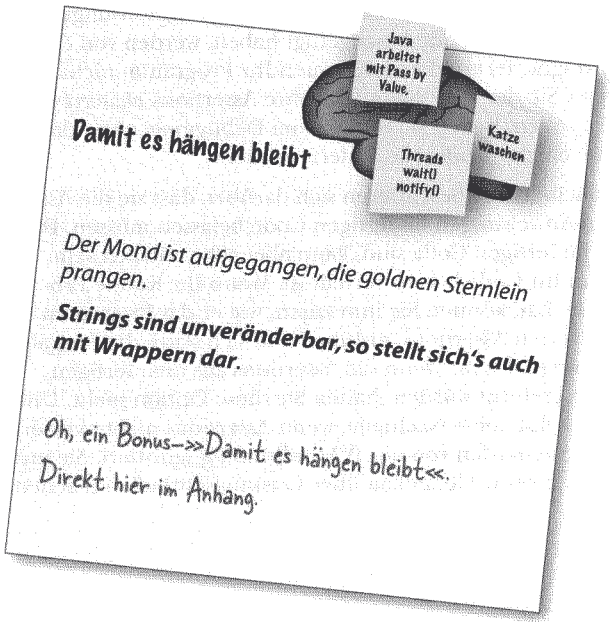
Im Math-Kapitel haben wir über die beiden Hauptverwendungszwecke der Wrapper-Klassen gesprochen:

- Einen elementaren Wert einpacken, damit er vorgeben kann, ein Objekt zu sein.
- Eine der statischen Hilfsmethoden (z.B. Integer.parse Int()) verwenden.

Es ist wichtig, dass Sie sich merken, dass die Sache für das Wrapper-Objekt gelaufen ist, wenn Sie es erzeugt haben:

```
Integer iWrap = new Integer(42);
```

Sein Wert wird *immer* 42 sein. ***Es gibt keine Setter-Methode für ein Wrapper-Objekt.*** Sie können *iWrap* natürlich auf ein *anderes* Wrapper-Objekt verweisen lassen, aber dann haben Sie *zwei* Objekte. Haben Sie ein Wrapper-Objekt erzeugt, gibt es keine Möglichkeit mehr, den *Wert* dieses Objekts zu ändern!



# Platz 8: Assertions

Wir haben nicht viel darüber gesagt, wie Sie Ihr Java-Programm debuggen, während Sie es entwickeln. Wir glauben, dass man Java auf der Kommandozeile lernen sollte, wie wir das in diesem Buch gemacht haben. Wenn Sie Java-Profi sind und sich entscheiden, mit einer IDE* zu arbeiten, stehen Ihnen vielleicht andere Debugging-Tools zur Verfügung. In den guten alten Zeiten pflanzte ein Programmierer einfach einen Haufen System.out.println( )-Anweisungen in seinen Code, wenn er ihn debuggen wollte. Durch sie ließ er die aktuellen Werte von Variablen und »Ich bin hier«-Meldungen ausgeben, mit denen er prüfen konnte, ob das Programm ordentlich funktionierte. (Im Fertiggericht-Code in Kapitel 6 wurden einige dieser print-Anweisungen zum Debugging im Code stehen gelassen.) Wenn das Programm dann funktionierte, ging er es durch und entfernte diese Anweisungen wieder. Das war mühsam und fehleranfällig. Aber seit Java 1.4 (und 5.0) ist das Debugging viel einfacher geworden. Die Antwort?

## Assertions

Assertions sind wie gedopte System.out.println( )-Anweisungen. Fügen Sie sie Ihrem Code hinzu, wie Sie es mit println-Anweisungen machen würden. Der Java-5.0-Compiler setzt voraus, dass Sie Quelldateien kompilieren, die 5.0-kompatibel sind, deswegen ist das Kompilieren mit Assertions seit Java 5.0 per Default aktiviert.

Zur Laufzeit tun Sie nichts. Die Assertion-Anweisungen, die Sie Ihrem Code hinzugefügt haben, werden von der JVM ignoriert und verlangsamen Ihr Programm nicht. Aber wenn Sie der JVM sagen, dass Ihre Assertions *aktiviert* werden sollen, helfen sie Ihnen beim Debugging, ohne dass Sie eine einzige Codezeile ändern müssen!

Manche Leute beschweren sich darüber, dass sie die Assertion-Anweisungen im fertigen Code belassen müssen. Dass sie im fertigen Code sind, kann aber sehr nützlich sein, wenn Ihr Code bereits verteilt ist. Wenn Ihr Kunde Probleme hat, können Sie ihm sagen, wie er das Programm mit aktivierten Assertions ausführt, und sich dann die Ausgabe zusenden lassen. Wenn die Assertions aus dem fertigen Code entfernt würden, hätten Sie diese Option nicht. Und es gibt fast keine Nachteile, wenn Assertions nicht aktiviert sind. Sie werden von der JVM vollständig ignoriert. Sie müssen sich keine Gedanken über Leistungseinbußen machen.

## Wie man Assertions zum Arbeiten bringt

Fügen Sie Ihrem Code eine Assertion-Anweisung hinzu, wenn Sie meinen, dass etwas *wahr sein muss*, zum Beispiel:

```
assert (höhe > 0);

// bei true wird die Programmausführung
// normal fortgesetzt

// bei false wird ein AssertionError
// ausgelöst
```

Sie können dem Stacktrace etwas mehr Informationen hinzufügen, indem Sie Folgendes sagen:

```
assert (höhe > 0) : "höhe = " +
höhe + " gewicht = " + gewicht;
```

Der Ausdruck nach dem Doppelpunkt kann ein beliebiger gültiger Java-Ausdruck sein, ***der zu einem Wert ungleich null aufgelöst wird.*** Aber – ganz gleich was Sie tun – ***erzeugen Sie keine Assertions, die den Zustand eines Objekts ändern können!*** Wenn Sie das tun, kann die Aktivierung von Assertions zur Laufzeit dazu führen, dass sich das Verhalten Ihres Programms ändert.

## Mit Assertions kompilieren und ausführen

Mit Assertions *kompilieren*:

```
javac SpielTestlauf.java
```

(Beachten Sie, dass keine Kommandozeilen-Optionen erforderlich sind.)

Mit Assertions *ausführen*:

```
java -ea SpielTestlauf
```

* IDE steht für Integrated Development Environment. Dazu zählen Werkzeuge wie Eclipse, Borlands JBuilder oder NetBeans (netbeans.org).

# Platz 7: Block-Geltung

In Kapitel 9 haben wir darüber gesprochen, dass lokale Variablen nur so lange leben, wie sich die Methoden, in denen sie deklariert wurden, auf dem Stack befinden. Aber einige Variablen können sogar ein noch *kürzeres* Leben haben. In Methoden erzeugen wir oft *Codeblöcke*. Wir haben das die ganze Zeit gemacht, ohne dabei explizit den Begriff *Block* zu verwenden. Üblicherweise tauchen Blöcke in Methoden auf und werden von geschweiften Klammern { } eingeschlossen. Verbreitete Beispiele für Codeblöcke, die Sie bereits kennen, sind Schleifen (*for*, *while*) und Bedingungsausdrücke (wie *if*-Anweisungen).

Sehen wir uns ein Beispiel an:

```
void machZeug() {          <- Anfang des Methodenblocks.

  int x = 0;               <- Eine lokale Variable, deren Geltungsbereich die
                              ganze Methode ist.

  for(int y = 0; y < 5; y++) {   <- Anfang eines for-Schleifenblocks, und die Geltung
                                    von y ist auf die for-Schleife beschränkt!

    x = x + y;             <- Kein Problem, x und y sind beide in Geltung.

  }                        <- Ende des for-Schleifenblocks.

  x = x * y;               <- Grrr! Lässt sich nicht kompilieren! Der Geltungsbereich von y
                              wurde hier bereits verlassen!
}                          <- Ende des Methodenblocks, auch x ist jetzt nicht mehr in Geltung.
```

In diesem Beispiel war *y* eine Blockvariable, die in einem Block deklariert wurde. Die Variable *y* verlor ihre Geltung, sobald die for-Schleife endete. Ihre Java-Programme lassen sich besser debuggen und erweitern, wenn Sie, wenn möglich, lokale Variablen statt Instanzvariablen verwenden und Blockvariablen statt Instanzvariablen. Der Compiler sichert dann, dass Sie die Variable nicht mehr verwenden, wenn der Geltungsbereich der Variablen verlassen wurde. Sie müssen sich also keine Gedanken mehr über Laufzeit-Kernschmelzen machen.

# Platz 6: Verkettete Aufrufe

Auch wenn Sie in diesem Buch so manches gesehen haben, haben wir trotzdem versucht, unsere Syntax so sauber und lesbar wie möglich zu halten. Es gibt in Java allerdings eine Menge legaler Abkürzungen, auf die Sie sicher stoßen werden, insbesondere wenn Sie viel Code lesen müssen, den Sie nicht selbst geschrieben haben. Ein verbreiteteres Konstrukt, das Ihnen begegnen wird, wird als *verkettete Aufrufe* bezeichnet, zum Beispiel:

```
StringBuilder sb = new StringBuilder("Wärme");
sb = sb.delete(3,5).insert(2,"inte").deleteCharAt(1);
System.out.println("sb = " + sb);
// Ergebnis ist sb = Winter
```

Was in aller Welt passiert in der zweiten Codezeile? Zugegeben, das ist ein konstruiertes Beispiel. Aber Sie müssen lernen, wie man so was entziffert.

1) Arbeiten Sie sich von links nach rechts vor.

2) Ermitteln Sie das Ergebnis des am weitesten links stehenden Methodenaufrufs, hier `sb.delete(3,5)`. Schlagen Sie das in der StringBuffer-API-Dokumentation nach, werden Sie sehen, dass die Methode `delete()` ein StringBuffer-Objekt zurückliefert. Das Ergebnis der Ausführung der Methode `delete()` ist ein StringBuffer-Objekt mit dem Wert »Wär«.

3) Die links als Nächstes kommende Methode (`insert()`) wird auf dem neu erzeugten StringBuffer-Objekt »Wär« aufgerufen. Das Ergebnis dieses Methodenaufrufs (der Methode `insert()`) ist *auch* ein StringBuffer-Objekt (auch wenn es nicht den gleichen Typ haben muss, wie der Rückgabewert der vorangehenden Methode). Und so kommt es, dass das zurückgelieferte Objekt verwendet werden kann, um die rechts als Nächstes kommende Methode aufzurufen. Theoretisch können Sie in einer einzigen Anweisung so viele Methoden verketten, wie Sie wollen (es passiert allerdings selten, dass Sie in einer Anweisung mehr als drei verkettete Methoden sehen). Ohne die Verkettung wäre die zweite Zeile des Codes lesbarer und sähe ungefähr so aus:

```
sb = sb.delete(3,5);
sb = sb.insert(2,"inte");
sb = sb.deleteCharAt(1);
```

Aber hier ist ein verbreiteteres und nützlicheres Beispiel. Sie haben bereits gesehen, wie wir es verwendet haben, aber wir dachten, wir sollten hier doch darauf hinweisen. Sie verwenden das, wenn Ihre main()-Methode eine Instanzmethode der Hauptklasse aufrufen muss und Sie keine *Referenz* auf eine Instanz der Klasse festhalten müssen. Anders gesagt: main() muss die Instanz *nur* erzeugen, damit main() eine der *Methoden* der Instanz aufrufen kann.

```
class Foo {
  public static void main(String [] args) [
    new Foo().los();
  }
  void los() {
    // hier steckt das, was wir EIGENTLICH suchen ...
  }
}
```

Wir möchten los() aufrufen, interessieren uns aber nicht für die Foo-Instanz und machen uns deswegen nicht die Mühe, das neue Foo-Objekt einer Referenz zuzuweisen.

# Platz 5: Anonyme und statische eingebettete Klassen

## Eingebettete Klassen gibt es in vielen Geschmacksrichtungen

Im GUI-Event-Handling-Teil dieses Buchs haben wir damit begonnen, innere (eingebettete) Klassen als Lösung für die Implementierung von Listener-Interfaces zu verwenden. Das ist die verbreitetste, praktischste und lesbarste Form einer inneren Klasse – wenn die Klasse einfach in die geschweiften Klammern einer anderen Klasse eingebettet wird. Und denken Sie daran: Das heißt, dass Sie eine Instanz der äußeren Klasse benötigen, um an eine Instanz der inneren Klasse zu kommen, weil die innere Klasse ein *Member* der äußeren/einschließenden Klasse ist.

Aber es gibt noch weitere Arten von eingebetteten Klassen. Dazu zählen *statische* und *anonyme* eingebettete Klassen. Wir werden das hier nicht in allen Einzelheiten betrachten, aber wir möchten nicht, dass Sie aus der Bahn geworfen werden, wenn Sie im Code anderer eine seltsame Syntax sehen. Denn von fast allem, was man mit der Programmiersprache Java machen kann, führt wahrscheinlich nichts zu bizarrer scheinendem Code als anonyme innere Klassen. Aber wir beginnen mit etwas Einfacherem – statischen eingebetteten Klassen.

## Statische eingebettete Klassen

Was statisch heißt, wissen Sie bereits – etwas, das an eine Klasse, nicht an eine bestimmte Instanz gebunden ist. Eine statische innere Klasse sieht im Prinzip genau so aus wie die nicht statischen Klassen, die wir für Event-Listener verwendet haben. Sie wird nur mit dem Schlüsselwort `static` markiert.

```
public class FooAußen {
  static class BarInnen {
    void sags() {
      System.out.println("Methode einer statischen eingebetteten Klasse");
    }
  }
}

class Test {
  public static void main (String[] args) {
    FooAußen.BarInnen foo = new FooAußen.BarInnen();
    foo.sags();
  }
}
```

Eine statische Klasse ist eben das – eine Klasse, die in eine andere Klasse eingeschlossen ist und mit dem static-Modifizierer markiert ist.

Weil eine statische Klasse statisch ist, brauchen Sie keine Instanz der äußeren Klasse. Sie verwenden einfach den Namen der Klasse – so wie Sie statische Methoden aufrufen oder auf statische Felder zugreifen.

Statische innere Klassen haben wieder eine gewisse Ähnlichkeit mit gewöhnlichen, nicht eingebetteten Klassen, weil sie keine besondere Beziehung zu dem äußeren Objekt haben, das sie einschließt. Aber weil statische eingebettete Klassen immer noch als *Member* der einschließenden/äußeren Klasse betrachtet werden, können Sie trotzdem auf private Member der äußeren Klasse zugreifen ... aber *nur auf die, die ebenfalls statisch sind.* Da die statische eingebettete Klasse nicht an eine Instanz der äußeren Klasse gebunden ist, hat sie keine Möglichkeit, auf nicht statische (instanzgebundene) Variablen und Methoden zuzugreifen.

# Platz 5: Anonyme und statische eingebettete Klassen (Fortsetzung)

## Der Unterschied zwischen eingebetteten und inneren Klassen

Jede Java-Klasse, die im Geltungsbereich einer anderen Klasse definiert wird, kann als ***eingebettete*** Klasse bezeichnet werden. Es spielt keine Rolle, ob die Klasse anonym, statisch, gewöhnlich oder sonst irgendwas ist. Wenn sie sich in einer anderen Klasse befindet, wird sie in technischer Hinsicht als *eingebettete* Klasse betrachtet. Aber *nicht statische* eingebettete Klassen werden oft als *innere* Klassen bezeichnet, und so haben auch wir das in vorangehenden Teilen dieses Buchs gemacht. Unterm Strich heißt das, dass alle inneren Klassen eingebettete Klasse sind, aber nicht alle eingebetteten Klassen sind auch innere Klassen.

## Anonyme innere Klassen

Stellen Sie sich vor, Sie schreiben GUI-Code, und plötzlich fällt Ihnen auf, dass Sie eine Instanz einer Klasse benötigen, die ActionListener implementiert. Aber Sie stellen fest, dass Sie keine Instanz von ActionListener *haben*. Und dann wird Ihnen klar, dass Sie nie eine *Klasse* für diesen Listener geschrieben haben. An diesem Punkt haben Sie zwei Möglichkeiten:

1) Sie können in Ihrem Code eine innere Klasse schreiben, wie wir das in unserem GUI-Code gemacht haben, diese dann instantiieren und diese Instanz an die Event-Registrierungsmethode (addActionListener()) des Buttons übergeben.

ODER

2) Erstellen Sie genau hier, vor Ort, eine *anonyme* innere Klasse und instantiieren Sie sie. ***Genau an der Stelle, an der Sie das Listener-Objekt benötigen.*** Sie haben richtig gehört: Sie erstellen die Klasse und erzeugen die Instanz genau an dem Ort, an dem Sie normalerweise nur die Instanz angeben würden. Denken Sie einen Augenblick darüber nach – das bedeutet, dass Sie dort, wo Sie eine *Instanz* als Methodenargument angeben, auch den vollständigen Body der zu instantiierenden *Klasse* angeben.

```
import java.awt.event.*;
import javax.swing.*;
public class TestAnon {
  public static void main (String[] args) {

    JFrame frame = new JFrame();
    JButton button = new JButton("click");
    frame.getContentPane().add(button);
    // button.addActionListener(quitListener);
```

Wir haben einen Frame gemacht und ihm einen Button hinzugefügt und müssen jetzt auf dem Button einen ActionListener registrieren. Aber wir haben nie eine Klasse erstellt, die das Interface ActionListener implementiert ...

Üblicherweise würden wir in etwa Folgendes machen – eine Referenz zu einer Instanz einer inneren Klasse übergeben ... einer inneren Klasse, die ActionListener (und die Methode actionPerformed()) implementiert.

Diese Anweisung

```
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ev) {
        System.exit(0);
      }
    });
```

endet hier!

```
  }
}
```

Beachten Sie, dass wir >>new ActionListener()<< sagen, obwohl ActionListener ein Interface ist und man eigentlich keine Instanz davon MACHEN kann! Aber diese Syntax besagt eigentlich: >>Erzeuge eine neue (namenlose) Klasse, die das Interface ActionListener implementiert, und nimm dazu die hier angegebene Implementierung der Interface-Methode actionPerformed().<<

Aber anstatt eine Objektreferenz anzugeben, geben wir hier die ganze Klassendefinition an!! Anders gesagt: Wir schreiben die Klasse, die ActionListener implementiert, GENAU DA WO WIR SIE BRAUCHEN. Und dann sagen wir new auf der hier und jetzt definierten Klasse und erhalten die ActionListener-Instanz, die wir brauchen.

# Platz 4: Zugriffsebenen und Zugriffsmodifizierer (wer sieht was?)

Java besitzt *vier* Zugriffs*ebenen* und *drei* Zugriffs*modifizierer*. Es gibt nur *drei* Modifizierer, weil *Default* (das Sie erhalten, wenn Sie keinen Zugriffsmodifizierer verwenden) eine der vier Zugriffsebenen *ist*.

**Zugriffsebenen** (in aufsteigender Reihenfolge der Beschränktheit)

*public* ← public bedeutet, dass jeder Code von überall auf das öffentliche Ding (mit Ding meinen wir Klasse, Methode, Variable, Konstruktor usw.) zugreifen kann.

*protected* ← protected funktioniert genau wie Default (Code im gleichen Package hat Zugriff), erlaubt ABER AUCH, dass Unterklassen, die sich nicht im gleichen Package befinden, das geschützte Ding erben.

*default* ← Default-Zugriff bedeutet, dass nur Code aus dem gleichen Package wie dieses Ding auf das Ding mit Default-Zugriff zugreifen kann.

*private* ← private bedeutet, dass nur Code aus dieser Klasse auf das private Ding zugreifen kann. Denken Sie daran, dass private privat für die Klasse heißt, nicht privat für das Objekt. Ein Hund kann das private Zeug in einem anderen Hund sehen, aber eine Katze kann das private Zeug eines Hunds nicht sehen.

### Zugriffsmodifizierer

```
public
protected
private
```

Meist verwenden Sie nur die Zugriffsmodifizierer public und private.

### public

Verwenden Sie public für Klassen, Konstanten (mit static final markierte Variablen) und Methoden, die Sie für anderen Code veröffentlichen (Getter und Setter beispielsweise), sowie die meisten Konstruktoren.

### private

Verwenden Sie private für fast alle Instanzvariablen und für Methoden, die äußerer Code nicht aufrufen können soll (d.h. Methoden, die von den öffentlichen Methoden Ihrer Klassen *verwendet* werden).

Aber auch wenn Sie die beiden anderen (protected und default) nicht verwenden, müssen Sie trotzdem wissen, was sie bewirken, weil Sie in anderem Code auf sie stoßen könnten.

# Platz 4: Zugriffsebenen und Zugriffsmodifizierer (Fortsetzung)

## default und protected

### default

Die Zugriffsebenen protected und default sind an Pakete gebunden. Default-Zugriff ist einfach – es bedeutet, dass nur Code *aus demselben Package* auf Code mit Default-Zugriff zugreifen kann. Auf eine Klasse mit Default-Zugriff (d.h. eine Klasse, die nicht explizit als *public* deklariert ist) kann also nur aus anderen Klassen aus dem gleichen Paket wie die Klasse mit Default-Zugriff zugegriffen werden.

Aber was heißt das überhaupt: auf eine Klasse *zugreifen*? Code, der keinen Zugriff auf eine Klasse hat, kann an diese Klasse noch nicht einmal *denken*. Und mit »Denken« meinen wir, die Klasse im Code *verwenden*. Wenn Sie auf Grund dieser Zugriffsbeschränkung keinen Zugriff auf eine Klasse haben, dürfen Sie die Klasse nicht instantiieren, ja Sie dürfen sie nicht einmal als Typ für eine Variable, ein Argument oder einen Rückgabetyp verwenden. Sie können sie in Ihrem Code überhaupt nicht verwenden! Wenn Sie es versuchen, beschwert sich der Compiler.

Denken Sie über die Implikationen nach – eine Default-Klasse mit öffentlichen Methoden heißt also, dass die Methoden eigentlich nicht wirklich öffentlich sind. Sie können auf keine Methode zugreifen, wenn Sie die Klasse nicht *sehen* können.

Warum würde jemand den Zugriff auf Code auf Paket-Ebene beschränken wollen? Üblicherweise sind Packages als eine Gruppe von Klassen entworfen, die als ein aufeinander bezogener Satz von Klassen zusammenarbeiten. Es könnte also sinnvoll sein, dass die Klassen auf den Code von Klassen im gleichen Package Zugriff haben, während das Package als Ganzes nur eine beschränkte Anzahl von Klassen und Methoden für die Außenwelt (d.h. Code außerhalb dieses Pakets) veröffentlicht.

Okay, das ist default. Es ist einfach – wenn etwas Default-Zugriff hat (was, Sie erinnern sich, heißt, dass es keinen expliziten Zugriffsmodifizierer gibt), kann nur Code aus dem gleichen Package wie das Default-*Ding* (Klasse, Variable, Methode, innere Klasse) auf das *Ding* zugreifen.

Wozu ist dann *protected* gut?

### protected

Geschützter Zugriff heißt fast das Gleiche wie Default-Zugriff. Es gibt nur eine Ausnahme: Er erlaubt Unterklassen, das geschützte Ding zu *erben, auch wenn sich diese Unterklassen außerhalb des Packages der Superklasse befinden, die sie erweitern*. Das ist es schon. Das ist *alles*, was protected Ihnen bringt – die Möglichkeit, Unterklassen außerhalb des Pakets der Superklasse zu haben, die trotzdem noch Teile der Klasse, einschließlich Methoden und Konstruktoren, *erben*.

Viele Entwickler finden wenig Gründe, protected zu verwenden. Aber in manchen Entwürfen wird es verwendet, und vielleicht stellen auch Sie eines Tages fest, dass es genau das ist, was Sie brauchen. Eines der interessanten Dinge an protected ist, dass es – anders als die anderen Zugriffsebenen – nur Auswirkungen auf die *Vererbung* hat. Wenn eine Unterklasse außerhalb des Packages eine *Referenz* auf eine Instanz der Superklasse hat (einer Superklasse, die beispielsweise eine geschützte Methode hat), kann die Unterklasse über die Superklassenreferenz nicht auf diese geschützte Methode zugreifen! Die Unterklasse kann nur auf die Methode zugreifen, indem sie sie *erbt*. Anders gesagt: Die Unterklasse-außerhalb-des-Packages hat keinen *Zugriff* auf die geschützte Methode, sie *hat* die Methode nur – durch Vererbung.

# Platz 3: String und StringBuffer/StringBuilder-Methoden

Zwei der am häufigsten verwendeten Klassen in der Java-API sind String und StringBuffer (denken Sie an Platz 9 ein paar Seiten weiter oben: Strings sind unveränderbar, StringBuffer/StringBuilder können also erheblich effizienter sein, wenn Sie Strings bearbeiten müssen). Ab Java 5.0 sollten Sie die Klasse String***Builder*** statt String***Buffer*** verwenden, es sei denn, Ihre String-Bearbeitungen müssen Thread-Safe sein, was nicht oft der Fall ist. Hier ist ein kurzer Überblick über die **wichtigsten** Methoden in diesen Klassen:

***Folgende Methoden haben String und StringBuffer/StringBuilder gemeinsam:***

```
char charAt(int index);                  // welches Zeichen befindet sich an dieser Position
int length();                            // wie lang ist das Ding
String substring(int start, int end);    // gib mir einen Teil von dem Ding
String toString();                       // was ist der String-Wert von dem Ding
```

***Strings verketten:***

```
String concat(String);                   // mit der Klasse String
StringBxxxx append(String);              // mit StringBuffer & StringBuilder
```

***Die Klasse String bietet außerdem:***

```
String replace(char old, char new); // alle Vorkommen des Zeichens ersetzen
char [] toCharArray();              // in ein Array von chars umwandeln
String toLowerCase();               // alle Zeichen in Kleinbuchstaben umwandeln
String toUpperCase();               // alle Zeichen in Großbuchstaben umwandeln
String trim();                      // Whitespace von den Enden entfernen
String valueOf(char [])             // aus einem char-Array einen String machen
String valueOf(int i)               // aus einem elementaren Wert einen String machen
                                    // die anderen elementaren Typen werden ebenfalls unterstützt
```

***Die Klassen StringBuffer & StringBuilder bieten außerdem:***

```
StringBxxxx delete(int start, int end);                          // einen Teil löschen
StringBxxxx insert(int offset, elementarer Typ oder char []);    // etwas einfügen
StringBxxxx replace(int start, int end, String s);               // diesen Teil durch diesen String ersetzen
StringBxxxx reverse();                                           // den SB umkehren
void setCharAt(int index, char ch);                              // das entsprechende Zeichen ersetzen
```

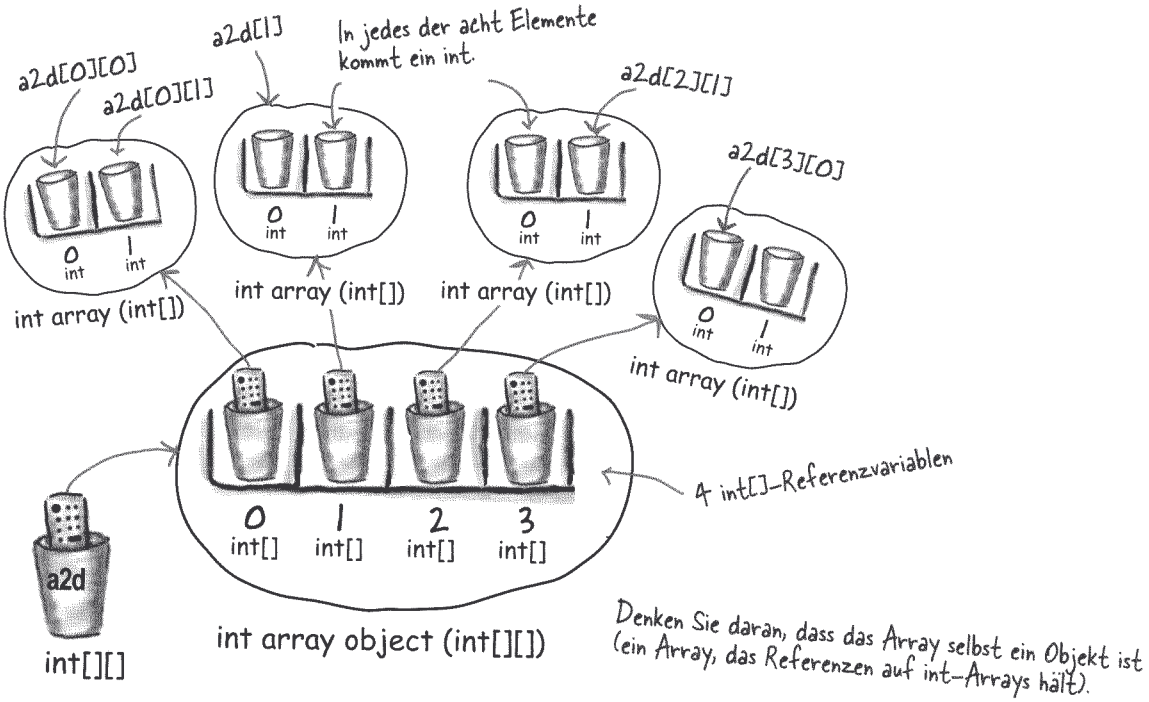
Hinweis: StringB***xxxx*** verweist auf String*Buffer* oder String*Builder*, je nachdem, was Sie verwenden.

# Platz 2: Mehrdimensionale Arrays

Wenn Sie beispielsweise ein zweidimensionales Array der Größe 4 x 2 erstellen, würden Sie sich in den meisten Sprachen ein Rechteck mit 4 Elementen mal 2 Elementen, also insgesamt 8 Elementen vorstellen. Aber in Java würde so ein Array tatsächlich aus *5* verknüpften Arrays bestehen! In Java ist ein zweidimensionales Array einfach *ein Array von Arrays*. (Ein dreidimensionales Array ist ein Array von Arrays von Arrays, aber damit herumzuspielen überlassen wir Ihnen.) Das funktioniert so:

```
int[][] a2d = new int [4][2];
```

Die JVM erzeugt ein Array mit 4 Elementen. *Jedes* dieser vier Elemente ist eigentlich eine Referenzvariable, die auf ein (neu erzeugtes) int-Array mit 2 Elementen verweist.




### Mit mehrdimensionalen Arrays arbeiten

- Auf das zweite Element im dritten Array zugreifen:

  ```
  int x = a2d[2][1]; // denken Sie daran, 0-basiert!
  ```

- Eine eindimensionale Referenz auf eins der Unter-Arrays erzeugen: `int[] kopie = a2d[1];`
- Abkürzung zur Initialisierung eines 2x3-Arrays: `int[][] x = { { 2,3,4 }, { 7,8,9 } };`
- Ein unregelmäßiges 2-D-Array erzeugen:

  ```
  int[][] y = new int [2][]; // macht nur das erste Array mit der Länge 2
  y[0] = new int [3]; // macht das erste Unter-Array mit der Länge 3
  y[1] = new int [5]; // macht das zweite Unter-Array mit der Länge 5
  ```

# Und nun Platz 1 der Top Ten-Themen, die es nicht ganz ins Buch geschafft haben ...

## Platz 1: Enumerationen (auch enumerierte Typen oder Enums genannt)

Wir haben über Konstanten gesprochen, die in der API definiert werden, **`JFrame.EXIT_ON_CLOSE`** beispielsweise. Eigene Konstanten können Sie erstellen, indem Sie die Variable als **`static final`** deklarieren. Aber manchmal möchten Sie einen Satz von konstanten Werten erzeugen, die die ***einzigen*** gültigen Werte für eine Variable repräsentieren. Dieser Satz gültiger Werte wird üblicherweise als *Enumeration* bezeichnet. Vor Java 5.0 konnte man in Java nur halbgare Enumerationen erstellen. Aber seit Java 5.0 können Sie vollwertige Enumerationen machen, um die Sie alle Ihre Freunde beneiden werden, die immer noch nicht mit Java 5.0 arbeiten.

**Wer darf mitspielen?**

Nehmen wir an, Sie erstellen eine Website für Ihre Lieblingsband und möchten sicherstellen, dass sich alle Kommentare auf ein bestimmtes Bandmitglied beziehen.

**So musste man früher ein »enum« simulieren:**

```
public static final int JERRY = 1;
public static final int BOBBY = 2;
public static final int PHIL = 3;

// später im Code

if (ausgewähltesBandmitglied == JERRY) {
  // JERRY-bezogenes Zeug erledigen
}
```

Wir hoffen, dass »ausgewähltesBandmitglied« einen zulässigen Wert hat, wenn wir diesen Punkt erreichen!

Das Gute an dieser Technik ist, dass sie dafür SORGT, dass der Code leichter zu lesen ist. Ein weiterer Vorteil ist, dass Sie den Wert der simulierten Enums, die Sie erzeugt haben, nie ändern können: `JERRY` bleibt immer `1`. Das Schlechte ist, dass es keine einfache oder saubere Möglichkeit gibt sicherzustellen, dass der Wert von `ausgewähltesBandmitglied` immer `1`, `2` oder `3` ist. Wenn irgendein nicht so leicht aufzuspürender Teil des Codes `ausgewähltesBandmitglied` auf `812` setzt, ist die Wahrscheinlichkeit hoch, dass das Ihren Code zum Einsturz bringt ...

# Platz 1: Enumerationen (Fortsetzung)

Die gleiche Situation unter Verwendung eines echten Java-5.0-Enums. Das ist zwar eine ziemlich einfache Enumeration, aber Enumerationen sind in der Regel *tatsächlich* so einfach.

**Ein neues, offizielles »Enum«:**

```
public enum Mitglieder { JERRY, BOBBY, PHIL };
public Mitglieder ausgewähltesBandmitglied;

// später im Code

if (ausgewähltesBandmitglied == Mitglieder.JERRY) {
  // JERRY-bezogenes Zeug machen
}
```

Sieht das nicht aus wie eine ganz gewöhnliche Klassendefinition? Es ist tatsächlich so, dass Enums eine besondere ART Klasse sind. Hier haben wir einen neuen Enum-Typ mit dem Namen >>Mitglieder<< erstellt.

Die Variable >>ausgewählteBandmitglieder<< hat den Typ >>Mitglieder<< und kann NUR einen der Werte >>JERRY<<, >>BOBBY<< oder >>PHIL<< annehmen.

Über den Wert dieser Variablen müssen Sie sich keine Gedanken mehr machen!

Mit dieser Syntax referenziert man eine Enum->>Instanz<<.

### Ihr Enum erweitert java.lang.Enum

Wenn Sie ein Enum erstellen, erzeugen Sie eine neue Klasse und ***erweitern implizit java.lang.Enum.*** Sie können ein Enum als eine eigenständige Klasse deklarieren in einer eigenen Quelldatei oder als Member einer anderen Klasse.

### Mit Enums »if« und »switch« verwenden

Mit dem gerade erstellten Enum können wir in unserem Code Verzweigungen durchführen, indem Sie eine `if-` oder `switch`-Anweisung nutzen. Beachten Sie auch, dass wir Enum-Instanzen mit `==` oder der Methode `.equals()` vergleichen können. Normalerweise gilt `==` als der bessere Stil.

```
Mitglieder n = Mitglieder.BOBBY;
if (n.equals(Mitglieder.JERRY)) System.out.println("Jerrrry!");
if (n == Mitglieder.BOBBY) System.out.println("Ratte");
```

Einen Enum-Wert einer Variablen zuweisen.

Das funktioniert beides wunderbar!

>>Ratte<< wird ausgegeben.

```
Mitglieder wennName = Mitglieder.PHIL;
switch (wennName) {
  case JERRY: System.out.print("Lass es singen ");
  case PHIL: System.out.print("Lass es raus ");
  case BOBBY: System.out.println("Cassidy! ");
}
```

Pop-Quiz! Was wird ausgegeben?

Antwort: Lass es raus Cassidy!

# Platz 1: Enumerationen, Abschluss

### Eine richtig komplexe Version eines ähnlichen Enums

Sie können Ihren Enums einen Haufen Dinge hinzufügen, Konstruktoren, Methoden, Variablen und etwas, das als konstantenspezifischer Klassen-Body bezeichnet wird. Das ist nicht sehr verbreitet, aber Sie könnten darauf stoßen:

```
public class EnumKomplex {

  enum Namen {
    JERRY("Lead-Gitarre") { public String singt() {
                            return "klagend"; }
                          },
    BOBBY("Rhythmus-Gitarre") { public String singt() {
                                return "heiser"; }
                              },

    PHIL("Bass");

    private String instrument;

    Namen(String instrument) {
      this.instrument = instrument;
    }
    public String getInstrument() {
      return this.instrument;
    }
    public String singt() {
      return "gelegentlich";
    }
  }

  public static void main(String [] args) {
    for (Namen n : Namen.values()) {
      System.out.print(n);
      System.out.print(", Instrument: "+ n.getInstrument());
      System.out.println(", singt: " + n.singt());
    }
  }
```

Das ist ein Argument, das dem unten deklarierten Konstruktor übergeben wird.

Das sind so genannte >>konstanten-spezifische Klassen-Bodies<<. Stellen Sie sich das einfach so vor, als würden diese die grundlegenden Enum-Methoden (hier die Methode >>singt()<<) überschreiben, wenn singt() auf einer Variablen mit dem Enum-Wert JERRY oder BOBBY aufgerufen wird.

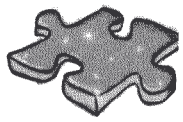
Das ist der Konstruktor des Enum. Er wird für jeden deklarierten Enum-Wert einmal ausgeführt (in diesem Fall dreimal).

Diese Methoden werden von >>main()<< aus aufgerufen.

Jedes Enum besitzt eine eingebaute >>values()<<-Methode, die, wie hier gezeigt, üblicherweise in einer >>for<<-Schleife verwendet wird.

```
Datei Bearbeiten Fenster Hilfe Raubkopie
%java EnumKomplex

JERRY, Instrument: Lead-Gitarre, singt: klagend
BOBBY, Instrument: Rhythmus-Gitarre, singt: heiser
PHIL, Instrument: Bass, singt: gelegentlich
%
```

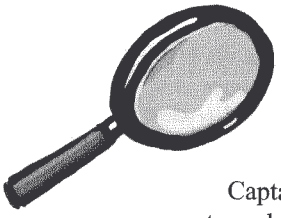
Beachten Sie, dass die elementare >>singt()<<-Methode nur aufgerufen wird, wenn der Enum-Wert keinen konstantenspezifischen Klassen-Body hat.



Kurzkrimi

## Ein langer Heimweg

Captain Byte vom Flächenland-Raumschiff »Große Überfahrt« hatte eine wichtige, streng geheime Übertragung von der Basis empfangen. Die Nachricht enthielt die 30 stark kodierten Navigationskodes, die die Große Überfahrt benötigte, um sich erfolgreich einen Weg durch die Feindsektoren zu bahnen. Feindliche Hackarianer aus einer Nachbargalaxie hatten eine teuflische Code-Mixer-Strahlung entwickelt, die auf dem Heap des Navigationscomputers der Großen Überfahrt Scheinobjekte generieren konnte. Außerdem konnte die Strahlung gültige Referenzvariablen so manipulieren, dass sie auf diese Scheinobjekte verwiesen. Der einzige Schutzmechanismus, den die Große Überfahrt gegen die bösartige Hackarianer-Strahlung besaß, war ein Inline-Virenprüfer, der in den hochmodernen Java-1.4-Code der Großen Überfahrt eingebettet werden konnte.



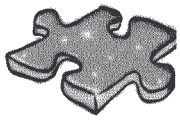
Captain Byte gab dem Gefreiten Schmidt die folgenden Programmieranweisungen zur Verarbeitung der kritischen Navigationskodes:

»Stecken Sie die ersten fünf Kodes in ein Array des Typs ParsecSchlüssel. Stecken Sie die letzten 25 Kodes in ein fünf mal fünf Elemente großes zweidimensionales Array des Typs QuadrantSchlüssel. Übergeben Sie die beiden Arrays an die Methode kursAnlegen() der finalen öffentlichen Klasse SchiffsNavigation. Wenn der Kurs zurückgeliefert wird, lassen Sie den Inline-Virenprüfer gegen alle Referenzvariablen des Programms laufen, führen dann das Programm NavSim aus und bringen mir die Ergebnisse.«

Ein paar Minuten später kam der Gefreite Schmidt mit der NavSim-Ausgabe zurück. »NavSim-Ausgabe zur Abnahme bereit, Captain«, rief der Gefreite Schmidt. »In Ordnung«, antwortete der Captain, »den Bericht zum Operationsablauf, bitte«. »Jawohl, Captain!«, antwortete der Gefreite. »Ich habe mit folgendem Code ein Array vom Typ ParsecSchlüssel deklariert und konstruiert: ParsecSchlüssel [] p = new ParsecSchlüssel[5]; und dann mit folgendem Code ein Array vom Typ QuadrantSchlüssel deklariert: QuadrantSchlüssel [] [] q = new QuadrantSchlüssel [5] [5];. Dann habe ich die ersten 5 Kodes mit einer for-Schleife ins ParsecSchlüssel-Array geladen und dann die anderen 25 Kodes mit geschachtelten for-Schleifen ins QuadrantSchlüssel-Array; anschließend Virenprüfer gegen alle 32 Referenzvariablen laufen gelassen, 1 für das ParsecSchlüssel-Array, 5 für seine Elemente, 1 für das QuadrantSchlüssel-Array, 25 für seine Elemente; nach negativer Virenprüfung NavSim-Programm ausgeführt und anschließend, sicherheitshalber, noch einmal Virenprüfer laufen lassen … Captain!»

Captain Byte ließ den Gefreiten lange unter seinem coolen Blick schmoren und sagte dann ruhig: »Stubenarrest, Schmidt. Wegen Gefährdung der Sicherheit des Schiffs. Ich will Ihr Gesicht erst wieder auf der Brücke sehen, wenn Sie ordentlich Java gelernt haben! Leutnant Boole, übernehmen Sie für den Gefreiten und machen Sie das ordentlich!«

Warum hat der Captain den Gefreiten zu Stubenarrest verdonnert?



# Kurzkrimi-Lösung

## Ein langer Heimweg

Captain Byte wusste, dass mehrdimensionale Arrays in Java eigentlich Arrays von Arrays sind. Es wären eigentlich insgesamt 31 Referenzvariablen erforderlich, um auf alle Komponenten des fünf mal fünf Felder großen QuadrantSchlüssel-Arrays q zuzugreifen:

1 – Referenzvariablen für q

5 – Referenzvariablen für q[0] - q[4]

25 – Referenzvariablen für q[0][0] - q[4][4]

Der Gefreite hatte die Referenzvariablen für die fünf eindimensionalen Arrays vergessen, die in das Array q eingebettet sind. Jede dieser Referenzvariablen hätte von der Hackarianer-Strahlung korrumpiert worden sein können, ohne dass die Prüfung des Gefreiten dieses Problem offen gelegt hätte.

# Index

## Symbole


&, &&, |. || (Boolesche Operatoren) 151, 660

&, <<, >>, >>>, ^, |, ~ (Bit-Operatoren) 660

. (Punkt-Operator) 36
  Referenz 54

+ (String-Verknüpfungsoperator) 17

++ – (Inkrement/Dekrement) 105, 115

<, <=, ==,!=, >, >= (Vergleichsoperatoren)
  11, 86, 114, 151


## A


Abfangen von Exceptions 326
  catch 338
  mehrere Exceptions abfangen 329–330, 332
  try 321

abstract
  Klasse 200–210
  Klassenmodifizierer 200

abstrakte Methoden
  deklarieren 203
  Klassenmodifizierer 667
  Methodenmodifizierer 81, 667
  Variablenmodifizierer 81, 667
  Zugriff und Vererbung 180

ActionListener-Interface 358–361

addActionListener() 359–361

Aeron™ 28

aktiver Typ 480, 484

Akzessoren und Mutatoren *siehe* Getter und Setter

Anhang A 649–658
  BeatBox, endgültiger Client 650
  BeatBox, endgültiger Server 657

Anhang B
  Assertions 662
  Bit-Manipulation 660
  Block-Geltung 663
  mehrdimensionale Arrays 670
  String- und StringBuffer-Methoden 669
  Unveränderbarkeit 661
  verknüpfte Aufrufe 664
  Zugriffsstufen und Modifizierer 667

Animation 382–385

Äpfel und Birnen 137

API 154–155, 158–160
  ArrayList 532
  Collections 558

Argumente
  Methoden 74, 76, 78
  polymorphe 187

ArrayList 132–138, 156, 208, 558
  API 532
  ArrayList<Object> 211–213
  Autoboxing 288–289
  Casting 229

Arrays
  deklarieren 59
  elementare Arrays 59
  erzeugen 60
  length-Attribut 17
  mehrdimensionale 670
  Objekt-Arrays 60, 83
  Überblick 17, 59, 135
  vs. ArrayList 134–137
  zuweisen 59

Arzt 169

Assertions 662

atomare Codeblöcke 510–512
  *siehe auch* Threads

Audio *siehe* Midi
aufgegebene Objekte *siehe* Garbage Collection
ausführbare JARs 585–586
  mit Packages 592–593
Auslösen von Exceptions 323–326
Ausweichen vor Exceptions 335
Autoboxing 288–291
  und Operatoren 291
  Zuweisungen und 291

# B

Badewanne 177
BeatBox 316, 347, 472
  speichern und wiederherstellen 462
  *siehe auch* Anhang A
Becher 51
Bela Fleck 30
Bier 14
Bit-Operatoren 660
Bit-Verschiebung 660
Block-Geltung 663
boolean 51
Boolesche Ausdrücke 11, 114
  logische 151
BorderLayout-Manager 370–371, 401, 407
BoxLayout-Manager 411
break-Anweisung 105
BufferedReader 454, 478
BufferedWriter 453
byte 51
Bytecode 2

# C

Calendar 303–305
  Methoden 305
catch-Blöcke 326, 338
  mehrere Exceptions abfangen 329–330, 332
char 51
Chat-Client 486
  mit Threads 518
Chat-Server (einfacher) 520
Checkbox (JCheckBox) 416
Client/Server 473
Code-Fertiggericht 112, 152–153, 520
Collections 137, 533
  API 558
  ArrayList 137
  ArrayList<Object> 211–213
  Collections.sort() 534, 539
  HashMap 533
  HashSet 533
  LinkedHashMap 533
  LinkedList 533
  List 557
  Map 557, 567
  parametrisierte Typen 137
  Set 557
  TreeSet 533
Collections.sort() 534, 539
  Comparator 551
  compare() 553
Comparable 547, 566
  compareTo()-Methode 549
  und TreeSet 566
Comparator 551, 566
  und TreeSet 566
compare() 553
compareTo() 549
Compiler 2
  java -d 590
  Überblick 18

# D

Damit es hängen bleibt 53, 87, 157, 179, 227, 278
Dateien
  File-Klasse 452
  lesen aus 441, 454
  Quellcode-Datei-Struktur 7
  schreiben in 432, 447
Datumswerte
  Calendar 303
  formatieren 301
  GregorianCalendar 303
  java.util.Date 303
  Methoden 305
Deadlock 516
Deadly Diamond of Death 223
Default-Wert 84
default-Zugriff 668
Deklarationen
  Exceptions 335–336
  Instanzvariablen 50
  Überblick 50
Deployment-Optionen 582, 608
deserialisierte Objekte 441
  *siehe auch* Serialisierung
DotCom versenken 96–112, 139–150
double 51

# E

E/A
  BufferedReader 454, 478
  BufferedWriter 453
  Deserialisierung 441
  FileInputStream 441
  FileOutputStream 432
  FileWriter 447
  InputStreamReader 478
  mit Sockets 478
  ObjectInputStream 441
  ObjectOutputStream 432, 437
  Puffer 453
  Serialisierung 432, 434–439, 446, 460
  Streams 433, 437
EJB 631
elementare Typen 53
  ==-Operator 86
  Autoboxing 288–289
  boolean 51
  byte 51
  char 51
  double 51
  float 51
  int 51
  short 51
  type 51
  Wertbereiche 51
elementare Typen umwandeln
  explizit 117
Ende des Buchs 648
Ente 277
  erzeugen 242
  Garbage Collection 261
endgültige BeatBox
  die GUI erstellen 418
  Musik mit Graphiken 386
  Sound abspielen 339
  *siehe auch* Anhang A
Enumerations 671–672
Enums 671–672
equals() 561
  Object-Klasse 209
  Überblick 209
Event-Handling 357–361
  Event-Objekt 361
  innere Klassen verwenden 379
  Listener-Interface 358–361
Event-Quelle 359–361
Exceptions
  auslösen 323–326
  ausweichen 335–336
  catch 321, 338

deklarieren 335–336
finally 327
Flusskontrolle 326
geprüfte vs. Laufzeit-Exceptions 324
Handeln oder Deklarieren-Gesetz 337
mehrere abfangen 329, 332
propagieren 335–336
remote Exceptions 616
try 321, 338
Überblick 320, 325, 338
Extreme Programming 101

# F

Fernsteuerung 54, 57
File 452
FileInputStream 441
*siehe auch* I/O
File-Klasse 452
FileOutputStream 432
FileReader 454
*siehe auch* I/O
FileWriter 447
finale
Klassen 189, 283
Methoden 189, 283
statische Variablen 282
Variablen 282–283
finally-Block 327
float 51
FlowLayout 403, 408–410
Flusskontrolle
Exceptions 326
Formatierung
Argumente 300
Datumswerte 301–302
Format-Spezifizierer 295–296
printf() 294
String.format() 294
Zahlen 294–295
for-Schleifen 105

# G

Garbage Collection
auswählbare Objekte 260–263
Heap 57–58
Referenzen neu zuweisen 58
Referenzen null-setzen 58
Überblick 40
Geisterstadt 109
Geltung
Variablen 236–238, 258–263
Generics 540, 542, 568–574
Jokerzeichen 574
Methoden 544
geprüfte Exceptions
Laufzeit-Exceptions vs. 324
geschweifte Klammern 10
Getter und Setter 79
Giraffe 50
Gleichheit 560
und hashCode() 561
Graphics2D-Klasse 366
Graphics-Objekt 364
Graphiken 364–366
*siehe auch* GUI
GregorianCalendar 303
GUI 406
Animation 382–385
BorderLayout 370–371, 401, 407
BoxLayout 403, 411
Buttons 405
Event-Handling 357–361, 379
FlowLayout 403, 408
Frames 400
Graphiken 363–367
ImageIcon-Klasse 365
JButton 400
JLabel 400
JPanel 400–401
JTextArea 414

JTextField 413
Komponenten 354, 363–368, 400
Layout-Manager 401–412
Listener-Interface 358–361
scrollen (JScrollPane) 414
Swing 354
Überblick 354, 400
GUI-Komponenten 354
JButton 354, 405
JCheckBox 416
JFrame 354, 400–401
JList 417
JPanel 400–401
JScrollPane 414, 417
JTextArea 414
JTextField 413
GUI-Konstanten
ALWAYS 415
NEVER 415
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_
ScrollPaneConstants.VERTICAL_SCROLLBAR_
GUI-Methoden
drawImage() 365
fillOval() 365
fillRect() 364
gradientPaint()
*siehe auch* GUI
paintComponent() 364
setColor() 364
setFont() 406


# H


HAS-A 177–181
hashCode() 561
HashMap 533, 558
HashSet 533, 558
Hashtable 558
Heap
Garbage Collection 40, 57, 58
Überblick 40, 57, 236–238


# I


I/O *siehe* E/A
if else 13
if-Anweisung 13
implements 224
import-Anweisung 155, 157
Importe
statische Importe 307
Initialisierung
elementare Typen 84
Instanzvariablen 84
statische Variablen 281
inkrementieren 105
innere Klassen
drei innere Klassen 381
Events 379
Überblick 376–386
InputStreamReader 478
Instantiierung *siehe* Objekte
Instanzvariablen
Default-Werte 84
deklarieren 84
initialisieren 84
Lebensdauer und Geltungsbereich 258–263
lokale Variablen vs. 236–239
statische Variablen vs. 277
Überblick 34, 73
int 50
elementare Typen 51
Integer *siehe* Wrapper
Interfaces
implementieren 224, 437
java.io.Serializable 437
mehrere implementieren 226
Überblick 219–227
zur Serialisierung 437
IP-Adresse *siehe* Netzwerk
IS-A 177–181, 251

# J

J2EE 631
JAR-Dateien
  ausführbare 585–586, 592
  ausführbare ausführen 586, 592
  grundlegende Befehle 593
  Hilfprogramm 593
  Manifest 585
  mit Java Web Start 598
Java in a Nutshell 158–159
Java Sound 317, 340
Java Web Start 597–601
Java, Überblick 5–6
javac *siehe* Compiler
Jini 632–635
JNLP 598
  jnlp-Datei 599
jnlp-Datei 598–599
Jokerzeichen 574
JPEG 365
JVM
  Überblick 2, 18
JWS *siehe* Java Web Start

# K

Kaffeebecher 51
Kamingespräche
  Überblick 18
Kaninchen 50
Kapselung
  Überblick 79–82
  Vorteile 80
Klassen
  abstrakte 200–210
  entwerfen 34, 41, 79
  finale 283
  konkrete 200–210
  vollständig qualifizierte Namen 154–155, 157
konkrete Klassen 200–210
Konstanten 282
Konstruktoren
  Superklassenkonstruktoren 250–256
  Überblick 240
  überladene 256
  verketten 250–256
Kontoführung *siehe* Rainer und Monika
Krafttraining 33, 167, 188
Kurzkrimi *siehe* Rätsel
Kurzschlussoperatoren, logische151

# L

Layout-Manager 401–412
  BorderLayout 370–371, 403, 407
  BoxLayout 403, 411
  FlowLayout 403, 408–410
Lerntipp 33, 108, 325
LinkedHashMap 533, 558
LinkedHashSet 558
LinkedList 533, 558
List 557
Listener
  Listener-Interface 358–361
literale Werte zuweisen
  elementare Typen 52
lokale Variablen 85, 236, 236–238, 258–263
long 51

# M

Mädchen in Badewanne 177
Mädchen, das es nicht versteht 182–188
main() 9, 38
Manifest-Datei 585
Map 557, 567

Math-Klasse
  Methoden 274–278, 286
  random() 111
mehrdimensionale Arrays 670
mehrere Threads *siehe* Threads
Mehrfachvererbung 223
Methoden
  abstrakte 203
  Argumente 74, 76, 78
  auf dem Stack 237
  finale 283
  generische Argumente 544
  Rückgabewerte 75, 78
  statische 274–278
  Überblick 34, 78
  überladen 191
  überschreiben 32, 167–192
Midi 317, 340–346, 387–390
Midi-Sequenzer 340–346
MINI Cooper 504
Modifizierer
  Klassen 200
  Methoden 203
Musik *siehe* Midi

# N

Namensgebung 53
  Klassen und Interfaces 154–155, 157
  Konflikte 587
  Packages 587
Netzwerk
  Ports 475
  Sockets 475
  Überblick 473
new 55
null-Referenzen 262

# O

Object-Klasse
  equals() 561
  hashCode() 561
  Überblick 208–216
ObjectOutputStream 432, 437
Objekte
  Arrays 59, 60, 83
  equals() 209, 561
  erzeugen 55, 240–256
  für Garbage Collection auswählbar 260–263
  Gleichheit 560
  Lebensdauer 258–263
  sperren 509
  Überblick 55
  vergleichen 209
Objektgraph 436, 438
Objektreferenzen 54, 56
  casten 216
  Gleichheit 560
  null setzen 262
  Polymorphie185–186
  vergleichen 86
  Zuweisung 55, 262
OO
  Deadly Diamond of Death 223
  Entwurf 34, 41, 79, 166–191
  HAS-A 177–181
  Interfaces 219–227
  IS-A 177–181, 251
  Polymorphie 183, 183–191, 206–217
  Superklassen 251–256
  überladen 191
  überschreiben 167–192
  Vererbung 166–192
  Verträge 190–191, 218
Operatoren
  Bedingungen 11
  Bit-Operatoren 660
  dekrementieren 115
  inkrementieren 105, 115

logische 151
und Autoboxing 291
Vergleichsoperatoren 151
Verschiebeoperatoren 660

# P

Packages 154–155, 157, 587–593
Code organisieren 589
Verzeichnisstruktur 589
paintComponent() 364–368
Parameter *siehe* Arguments
parametrisierte Typen 137
Parsen eines Ints *siehe* Wrapper
Parsen von Text mit String.split() 458
Pass-by-Copy *siehe* Pass-by-Value
Pass-by-value 77
Phras-O-Mat 16
Polymorphie 183–191
abstrakte Klassen 206–217
Argumente und Rückgabetypen 187
Referenzen des Typs Object 211–213
und Exceptions 330
Pool-Puzzle *siehe* Rätsel
Ports 475
printf() 294
PrintWriter 479
private-Zugriffsmodifizierer 81
Problem der verlorenen Aktualisierung *siehe* Threads
protected 668
public-Zugriffsmodifizierer 81, 668
Puffer 453–454
Punkt-Operator
Referenzen 54

# Q

Quellcode-Dateien, Struktur 7
Quizkarten-Generator 448–451

# R

Rainer und Monika 505–506
random() 111
Ratespiel 38
Rätsel
Java-Kreuzworträtsel 22, 120, 162, 350, 426, 603
Kurzkrimi 92, 527, 674
Pool-Puzzle 24, 44, 65, 91, 194, 232, 396
Referenzvariablen *siehe* Objektreferenzen
umwandeln 216
Registrierung, RMI 615, 617, 620
Remote-Interface *siehe* RMI
reservierte Wörter 53
riskanter Code 319–336
RMI
Client 620, 622
Compiler 618
Jini *siehe* Jini
Naming.lookup() 620
Registrierung 615, 617, 620
Remote-Exceptions 616
Remote-Implementierungen 615, 617
Remote-Interface 615–616
rmic 618
Skeleton 618
Stub 618
Überblick 614–622
UnicastRemoteObject 617
Universaldienstbrowser 636–648
rmic *siehe* RMI
Rückgabetypen
polymorphe 187
Überblick 75
Werte 78
run()in Runnable-Interface überschreiben 494
Runnable-Interface 492
lauffähig, Threadstatus 495
run() 493–494
Threads 493
Überblick 493

# S


Scheduling von Threads 496–498

schlafende Threads 501–503

Schleifen
- break 105
- for 105
- Überblick 10
- while 115

Schlüsselwörter 53

schreckliche Objekte 200

Schriften 406

Scrollen (JScrollPane) 414

Serialisierung 434–439, 446
- Deserialisierung 460
- Interface 437
- lesen *siehe* I/O
- ObjectInputStream *siehe* I/O
- ObjectOutputStream 432
- Objekte 460
- Objektgraph 436
- schreiben 432
- SerialVersionUID 461
- speichern 432
- transient-Schlüsselwort 439
- Versionierung 460–461
- wiederherstellen 460 *siehe auch* I/O

Server
- Servlet 625–627
- Socket 483 *siehe auch* Socket

Set 557
- Bedeutung von equals() 561
- Bedeutung von hashCode() 561

short 51

Skeleton *siehe* RMI

sleep() 501–503

Snowboard 214

Sockets
- Adressen 475
- erzeugen 478
- I/O 478
- Lesen aus 478
- Ports 475
- Schreiben in 479
- Server 483
- TCP/IP 475
- Überblick 475

Sortieren
- Collections.sort() 534, 539, 547
- Comparable-Interface 547, 549
- Comparator 551, 553
- TreeSet 564–566

Speicher
- Garbage Collection 260–263

Sperren
- Objekte 509
- Threads 509

Spezifizierer
- Argumentspezifizierer 300
- Formatspezifizierer 295, 298

sprechender Kopf 203

Stack
- Geltung 236
- Heap vs. 236
- Methoden auf 237
- Stacktrace 323
- Threads 490
- enumerierte Typen 671
- Initialisierer 282
- Methoden 274–278
- Methoden der Klasse Math 274–278
- statische Importe 307
- Variablen 282

Streams 433 *siehe auch* E/A

StringBuffer/StringBuilder
- Methoden 669

Strings
- Arrays 17
- Methoden 669
- parsen 458
- String.format() 294–297
- String.split() 458
- verketten 17

Stub *siehe* RMI
Stuhlkriege 28, 166
super 228
Superklasse 31, 166–192, 214–217, 228
Super-Konstruktor 250–256
Swing *siehe* GUI
Synchronisierung
  Methoden 510 *siehe auch* Threads
Syntax-Überblick 10, 12
System.out.print() 13
System.out.println() 13

# T

tanzendes Mädchen
TCP-Ports 475
Telluride 30
Testausdrücke 10, 11, 13
Testen und Extreme Programming 101
Text
  aus Datei lesen *siehe auch* E/A
  in Datei schreiben 447
  parsen mit String.split() 458
Textbereich (JTextArea) 414
Textfeld (JTextField) 413
Thread.sleep() 501–503
Threads
  Deadlock 516
  Problem der verlorenen Aktualisierung 512–514
  Rainer und Monika-Problem 505–507
  run() 493, 494
  Runnable 492–494
  Scheduling 496–498
  sleep() 501–503
  Sperren 509
  Stack 490–491
  start() 492
  starten 492
  synchronisierte 510–512
  Überblick 489–515
  Unvorhersagbarkeit 498–499
  Zusammenfassung 500, 517
  Zustände 495–496
throws 323–326
TippDesTages-Client 480
TippDesTages-Server 484
transient 439
träumendes Mädchen
  innere Klassen 375
  Java Web Start 596
TreeMap 558
TreeSet 533, 558, 564–566
try-Blöcke 321, 326
Typ 50
  Parameter 137, 542, 544
Typsicherheit 540
Typumwandlungen
  explizite, bei elementaren Typen 117
  explizite, bei Referenztypen 216
  implizite, bei elementaren Typen 117

# U

überladen 191
  Konstruktoren 256
überschreiben
  Polymorphie *siehe* Polymorphie
  Überblick 32, 167–192
  Methoden 563
Übungen
  Code-Magneten 20, 43, 64, 119, 312, 349, 467, 524–525
  Honigtopf 267
  Spielen Sie... 88, 118, 266, 310, 395
  Wahr oder Falsch 311, 348, 466, 602
  Welcher Layout-Manager? 424
  Wer bin ich 45, 89, 394
  Wie sieht das Bild aus? 230
  Wie sieht die Deklaration aus? 231

Universal-Dienst-Browser 636–648
Unterklassen 31, 166–192
unterschiedlich bellen 73
Unveränderbarkeit, Strings 661

# V

Variablen
  deklarieren 50, 54, 84, 236–238
  elementare 51–52
  Geltung 236–238
  lokale 85, 236–238
  null setzen 262
  Referenzen 54–56, 185–186
  statische *siehe* static
  zuweisen 52, 262
Variablendeklarationen 50
  elementare Variablen 51
  Instanzvariablen 84
  Referenzvariablen 54
Vererbung
  IS-A 214, 251
  super 228
  Tiere 170–175
  Überblick 31, 166–192
  und abstrakte Klassen 201
Vergleiche mit == 86
Verhalten 73
Verketten 17
verknüpfte Aufrufe 664
Verträge 190–191, 218
Verzeichnisstrukturen
  Packages 589
  Servlets 626
virtuelle Methodenaufrufe 175
vollständig qualifizierte Namen 154, 157
  Packages 587
Vorcode 99–102

# W

Wäsche, Exceptions 329
Web Start *siehe* Java Web Start
Wein 202
while-Schleifen 11, 115
Wrapper 287
  Autoboxing 288–289
  Integer.parseInt() 104, 106, 117
  Umwandlungswerkzeuge 292

# Z

Zahlen formatieren 294–295
Zuweisung von elementaren Typen 52
Zuweisung von Referenztypen 55, 57, 83

# Aber es muss kein Abschied sein

Sie kennen unsere Website nicht? Da haben wir Antworten für stumpfe Bleistifte, Beispiele, Code-Küchen, Code-Fertiggerichte und tägliche neue Beiträge im HeadFirst-Autoren-Blog!

**Schauen Sie doch einfach bei *wickedlysmart.com* vorbei!**